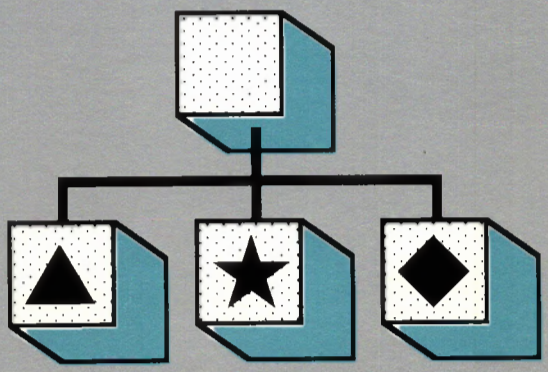
TURBO PASCAL

Volume 2

Data Input/Output

Object Professional™

A library of over 100 powerful object types containing over 1000 methods. Window object types let you use multiple overlapping resizeable windows providing: ■ Menus ■ Pick lists ■ Data entry ■ Text editing and much more. Data object types and system-oriented routines are included too!



TURBO Power

Object Professional

User's Manual

Volume 2
Data Input/Output

Copyright (c) 1988-1991, TurboPower Software
All Rights Reserved

Second Edition December 1991

P.O. Box 49009
Colorado Springs, CO 80949-9009

Technical Support: 719-260-6641
Fax: 719-260-7151
Sales: 800-333-4160
9 a.m. to 5 p.m. Mountain Time, Monday-Friday

CompuServe Mail: 76004,2611
CompuServe Support: PCVENB Section 6

Trademarks Mentioned

TurboPower Software and the distinctive TurboPower logo are trademarks of TurboPower Software.

Object Professional is a trademark of TurboPower Software.

Turbo Professional is a registered trademark of Sunny Hill Software, used under license to TurboPower Software.

Turbo Pascal, Turbo Assembler, Turbo Debugger, Turbo Database Toolbox, Turbo Editor Toolbox, SideKick, SideKickPlus, and SuperKey are trademarks or registered trademarks of Borland International.

DesqView is a trademark of Quarterdeck Office Systems.

Double DOS is a trademark of SoftLogic Solutions.

OmniView is a trademark of Sunny Hill Software.

Periscope is a trademark of The Periscope Company.

Hercules is a registered trademark of Hercules Computer Technology, Inc.

dBase is a trademark of Ashton-Tate.

Lotus is a trademark of Lotus Development Corporation.

WordStar is a registered trademark of MicroPro International Corporation.

MS-DOS, OS/2, Windows, CodeView, and Macro Assembler are trademarks of Microsoft Corporation.

IBM PC, XT, AT, PS/2, and TopView are registered trademarks of International Business Machines Corporation.

Table of Contents

Volume 1: Screen Management

1. Introduction 1-1
   - A. System Requirements 1-4
   - B. Files Supplied 1-5
   - C. Installation 1-7
   - D. Organization of the Manual 1-14
   - E. Getting Help with POPHELP 1-17
   - F. DESKPOP: Desktop Manager 1-24
   - G. Purchase Agreement 1-45

2. Principles of Organization 2-1

3. Screen and Keyboard 3-1
   - A. Overview 3-1
   - B. OPCRT: Enhanced CRT Unit 3-2
   - C. OPCOLOR : Color Names 3-49
   - D. OPMOUSE: Mouse Routines 3-52
   - E. OPCMD: Command Processing 3-82

4. Window Objects 4-1
   - A. Overview 4-1
   - B. OPFRAME: Frame Objects 4-8
   - C. OPWINDOW: Window Objects 4-55
   - D. OPPICK: Pick Lists 4-186
   - E. OPDIR: Directory and Path Lists 4-241
   - F. OPMENU: Menu Systems 4-297

Index IN-1

Volume 2: Data Input/Output

4. Window Objects (continued) 4-361
   - G. OPHELP: Help Systems 4-361
   - H. Utility and Sample Programs 4-420

5. Text Editors and File Browsers 5-1
   - A. Overview 5-1
   - B. OPMEMO: Basic Text Editor 5-2
   - C. OPEDITOR: Advanced Text Editor 5-30
   - D. OPBROWSE: File Browser 5-56
   - C. Sample Programs 5-79

6. Data Entry 6-1
A. Overview 6-1
B. OPABSFLD: Abstract Field Definitions 6-4
C. OPFIELD: Data Entry Fields 6-15
D. OPFEDIT: Field Editors 6-52
E. OPSELECT: Field Selection 6-55
F. OPENTRY: Data Entry Screens 6-110
G. OPEDIT: Advanced Line Editor 6-176
H. OPSEDIT: Simple Line Editor 6-197
I. OPDATE: Date and Time Variables 6-206
J. Utility and Sample Programs 6-228

7. Printing 7-1
A. Overview 7-1
B. OPPRNLOW: Physical Level Printer Support 7-3
C. OPPRINT: Logical Level Printer Support 7-27
D. OPDEVICE: Hardware Specific Printer Support 7-57
E. OPFORM: Printed Forms 7-66
F. Sample Programs 7-113

Index IN-1

Volume 3: TSRs and More

8. Data Objects 8-1
A. Overview 8-1
B. OPROOT: Object-Oriented Data Structures 8-2
C. OPLARRAY: Object-Oriented Large Arrays 8-154

9. String Handling 9-1
A. Overview 9-1
B. OPSTRING: String Manipulation 9-2
C. OPASCIIZ: Long ASCIIZ Strings 9-35
D. OPSTRDEV: Formatted Strings 9-49

10. DOS/BIOS Calls 10-1
A. Overview 10-1
B. OPDOS: Assorted DOS/BIOS Calls 10-3
C. OPEXEC: Swapping Exec Manager 10-32
D. OPENHKBD: Enhanced Keyboard Interface 10-42

11. Interrupts and TSRs 11-1
A. Overview 11-1
B. OPINT: ISR Management 11-4
C. OPTSR: TSR Management 11-13
D. Overview of Swappable TSRs 11-34
E. OPSWAP1: Swappable TSRs, Part 1 11-47
F. OPSWAP: Swappable TSRs, Part 2 11-56
G. OP8087: 8087 Support 11-58
H. Sample Programs 11-64

12. Memory Allocation 12-1
A. Overview 12-1
B. OPEMS: EMS Memory Management 12-3
C. OPEXTMEM: Extended Memory Management 12-12

13. Macros 13-1
A. Overview 13-1
B. OPMACRO: Keyboard Macros 13-3
C. OPMACED: Macro Editor 13-15
D. OPREPLAY: Macro Playback 13-18
E. Sample Programs 13-23

14. Miscellaneous 14-1
A. Overview 14-1
B. OPBCD: High-precision Reals 14-2
C. OPCLONE: Program Customization 14-16
D. OPERRHAN: Runtime Error Recovery 14-25
E. OPINLINE: Assorted Inline Macros 14-28
F. OPSORT: In-memory Sort Routine 14-45

15. Appendices 15-1
A. What's New in Object Professional 1.0 15-1
B. Unit Dependencies 15-4
C. Object Hierarchy 15-7
D. Error Codes 15-10
E. Using Overlays with Object Professional 15-14
F. Converting Turbo Professional Applications 15-16
G. Compiling Large Programs 15-24
H. For Further Reading 15-26

Procedure and Function Reference PR-1

Index IN-1

Object Quick Reference

Volume 1: Screen Management

3. Screen and Keyboard

ColorSet 3-48
CommandProcessor 3-95
CommandPacker 3-115

4. Window Objects

ScreenRect 4-10
AbstractFrame 4-31
Frame 4-42
RawWindow 4-62
AbstractWindow 4-69
StackWindow 4-118
CommandWindow 4-143
WindowStack 4-132
LoadableColorSet 4-162
PackedWindow 4-166
VirtScreen 4-175
PickList 4-208
DirList 4-251
PathList 4-285
MenuItemNode 4-314
Menu 4-323

Volume 2: Data Input/Output

4. Window Objects (continued)

AbstractHelpWindow 4-384
ScrollingHelpWindow 4-415

5. Text Editors and File Browsers

Memo 5-10
MemoFile 5-22
TextEditor 5-41
Browser 5-64

6. Data Entry
SelectField 6-18
EntryField 6-64
AbstractSelector 6-64
Selector 6-90
ScrollingSelector 6-103
EntryScreen 6-144
ScrollingEntryScreen 6-169
LineEditor 6-188
SimpleLineEditor 6-201

7. Printing
BasePrinter 7-10
DosPrinter 7-19
BiosPrinter 7-20
Printer 7-38
PrintField 7-88
PrintedForm 7-92
Report 7-109

Volume 3: TSRs and More

8. Data Objects
PointerStack 8-9
StaticQueue 8-21
SingleListNode 8-33
SingleList 8-34
DoubleListNode 8-43
DoubleList 8-44
CircularList 8-49
BitSet 8-52
LargeBitSet 8-64
StringDict 8-71
StringSet 8-80
StringArray 8-82
Root 8-99
IdStream 8-100
DosIdStream 8-120
BufIdStream 8-123
MemIdStream 8-125
DirEntry 8-132
Library 8-137
MemLibrary 8-148
AbstractArray 8-161

14. Miscellaneous
Cloner 14-19

G. OPHELP - Help Systems

If you've ever developed a context-sensitive help system for an application, you know that it takes substantial effort to write one that is efficient and easy to use. The OPHELP unit offers a powerful, convenient method to incorporate help screens into your programs.

There are three, perhaps four, steps required to develop a help system.

1. Write the help text
2. Compile the text into a compressed, indexed help file
3. Call the appropriate OPHELP methods to manage a help window
4. Optionally, use the CommandProcessor help hooks to connect the help window to your other window objects

To write the help text, you use any text editor, such as the one in Turbo's integrated development environment. Each help file contains as many topics as you need; OPHELP accesses each topic quickly by name or number. We'll describe the format of the text file in a moment. To compile the text, you use the supplied MAKEHELP utility.

OPHELP provides three objects to manage help windows. One of them, AbstractHelpWindow, is an abstract parent that provides the methods to access the binary help file but doesn't implement the methods to display the information. The other two, PagedHelpWindow and ScrollingHelpWindow, offer two different ways to display the help information and to interact with the user. The PagedHelpWindow works like the TPHELP unit in Turbo Professional: help is displayed a page at a time; the PgDn and PgUp keys move to different pages; the arrow keys move the highlight bar from one cross-reference link to the next. By contrast, ScrollingHelpWindow offers line-by-line vertical scrolling and horizontal scrolling; the hardware cursor remains visible and moves with the arrow keys to select cross-reference links. ScrollingHelpWindow is better suited to modern environments that support resizeable, zoomable windows; we've chosen to use it in the POPHELP popup reference guide.

If you like, the compressed help file can be linked directly into the target program, so that no additional files are required by the end user of the application. For hefty help files or large applications, the file is kept separate and OPHELP reads pieces of it into memory on demand.

The help system supports up to 10,000 help topics in one file. Each help topic may contain cross-references to other topics, with up to 50 cross-references for each topic. (These limits can be increased by modifying constants in OPHELP.) Help is displayed in a pop-up window through which the user may browse. Formatting controls allow help to appear in up to six different video attributes. The help compiler optionally word-wraps and paginates the original help text, under the control of directives that you insert.

You can choose which topic to display by using any of several techniques: directly by topic number, by topic name, by scanning context words from the current screen, by selecting from a master index, by cross-referencing to related items, or by referring to a stack of previously selected help items. Many of these options are invoked automatically with no further effort on your part. The OPHELP unit fully supports the mouse as well, leading to even friendlier help systems.

The text files used as input to the OPHELP system are downwardly compatible with those used for Turbo Professional. There are a few new directives in OPHELP, but all the old ones work just the same. **Object Professional's** MAKEHELP provides improved compression and some other new features; you'll need to recompile old help files before they can be used.

The remaining sections of this introduction provide an example you can run, describe how to develop help text files, and show how to compile them with MAKEHELP. The information applies to help systems developed for either the PagedHelpWindow or the ScrollingHelpWindow object, which use the same input files. Note, however, that the formatting of the file can be optimized for improved appearance in one object or the other.

After this introductory material, we'll continue with three sections covering the AbstractHelpWindow, PagedHelpWindow, and ScrollingHelpWindow objects, and then wrap up by describing a few procedures that OPHELP provides to detect which kind of help window is preferred for a particular help file.

A Simple Help Example

The following short program shows how to pick a help topic from the master index, display the help for that topic, and interpret the exit codes from the Process method. It uses the PagedHelpWindow, but the same methodology applies to the ScrollingHelpWindow. If you wish to run the program, note that ENTRY.HLP is the same help file used with the ENTRY.EXE example program. ENTRY.TXT is also supplied; you may wish to refer to it during the discussions that follow.

```
program HelpExample;  {EXHELP1.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpMouse, OpFrame, OpWindow, OpPick, OpHelp;
var
  PH : PagedHelpWindow;
  Finished : Boolean;
begin
  {Make a PagedHelpWindow with custom window options}
  if not PH.InitCustom(9, 8, 72, 18,                    {Window coordinates}
                       DefaultColorSet,                 {ColorSet}
                       DefWindowOptions or wBordered,   {Window options}
                       'ENTRY.HLP',                     {Help file}
                       PickVertical)                    {Pick Orientation}
  then begin
    WriteLn('Failed to Init PagedHelpWindow,  Status = ', InitStatus);
    Halt;
  end;

  {Add some features}
  PH.EnableExplosions(10);
  PH.wFrame.AddHeader(' Topic Index ', heTC);
  PH.AddMoreHeader(' || for more ', heBR, #24, #25, '', 2, 3, 0);
  PH.AddTopicHeader(1, 60, heTC);
  PH.AddMoreHelpHeader(
    ' PgUp/PgDn for more ', heBR, 'PgUp', 'PgDn', '/', 2, 7, 6);
  PH.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  PH.hwFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);

  {Loop through help system}
  Finished := False;
  repeat
    {Display master index and pick a help topic}
    PH.SetTopic(PickTopic);
    PH.Process;
    case PH.GetLastCommand of
      ccError,
      ccQuit :
        Finished := True;
```

```
        ccSelect :
          begin
            PH.SetTopic(PH.GetTopicChoice);
            repeat
              PH.Process;
              case PH.GetLastCommand of
                ccError :
                  Finished := True;
                ccQuit :
                  if not PH.InHelpMode then
                    {Escaped from pick window within help}
                    Finished := True;
                {...other Help or PickList exit commands...}
              end;
            until Finished or (PH.GetLastCommand = ccQuit);
          end;
        {...other PickList exit commands...}
      end;
    until Finished;

    PH.Done;
  end.
```

PH.InitCustom defines the help window's active coordinates, the colors to use, the window options, the name of the compiled help file, and an orientation for the master help index's pick list. Like any other window object, we can add headers and enable other special effects. Note that AddMoreHeader is a method inherited from the PickList, which is an ancestor of the PagedHelpWindow. This header appears only when the help window is in "pick mode," a term defined to mean that the master topic index is visible, in contrast to "help mode," which applies when the actual help text shows. The help window adds two more specialized header types, AddTopicHeader--which displays the current topic name while the window is in help mode--and AddMoreHelpHeader--which is like AddMoreHeader but indicates that more help text is available off-screen.

Similarly, we can add scroll bars, but with a new twist. Windows normally have two Frame objects associated with them: wFrame and aFrame (see OPWINDOW). The help window adds a third: hwFrame. The hwFrame object is displayed only when the window is in help mode, while the wFrame object is displayed when the window is in pick mode. By having two separate frames, we can use specialized headers that apply only to each one. In some cases, as with the scroll bars here, we must duplicate frame features to assure that they appear in both window modes.

In case you're wondering, it's possible to disable the help window's pick index and use it in a non-schizophrenic mode. You can do so dynamicallly, by clearing the hwPickIndex help option, or by not indexing any topics, as shown in the discussion of "Help File Directives" immediately following.

The Process loop again points out the dual nature of the help window. When the example passes the special topic number PickTopic (which equals zero) to SetTopic, Process switches the window into pick mode and displays the master topic list. If an error occurs or the user presses <Esc>, we set Finished to True to halt the example. Otherwise, the example sets the selected topic to the one returned by GetTopicChoice and starts another Process loop, which interprets exit commands somewhat differently. In this loop, if the user presses <Esc> and the window is in help mode, the inner loop exits and control returns to the outer loop to pick a new topic. If the user presses <Esc> while the window is in pick mode, the example is done.

How is it possible to get into pick mode within the inner loop? When help mode is active, the command processor interprets the <F1> key as a request to display the master help index. When this key is pressed, the Process method does not exit in order to switch into pick mode.

These nested loops may be more complex than your application will require. When you already know the topic you want (based on the context of the situation) and you haven't defined special help window exit commands, the loops disappear altogether:

```
PH.SetTopic(TopicYouWant);
PH.Process;
if PH.GetLastCommand = ccError then
  {Couldn't access help--beep or display message};
```

Help File Directives

The information displayed by the help system starts out as a text file that you must write. You put directives in that text file to tell the help compiler how to format and interpret it.

We typically use the extension .TXT to indicate a help text file, and reserve the extension .HLP for the binary file produced by MAKEHELP. Although the help system does not require these extensions, MAKEHELP uses them by default.

Directives in the TXT file are denoted by an exclamation point in the first column of the line, immediately followed by the directive name, followed by optional modifiers separated by one or more spaces. Only the first five characters of each directive name are significant.

Following is a description of the directives accepted by the help compiler. Refer to the file ENTRY.TXT for a real-world example of using these directives.

!WIDTH Width

Specifies the number of columns to be occupied by the active portion of the help window. By default, when the help is displayed, it will be separated from the edge of the active window by one column on both sides. Hence, the actual width for text is (Width-2) columns. When word-wrapping is activated, MAKEHELP wraps after (Width-2) columns. If no !WIDTH directive is specified, MAKEHELP assumes Width=40.

Because MAKEHELP word-wraps the text, the width specification for a help system is built into the help file. The word-wrap cannot be redone while the help is being displayed. However, the width of the visible help window may be chosen independently of the wrap width; this especially makes sense when displaying help with the ScrollingHelpWindow object.

MAKEHELP does not automatically insert page breaks into the help text, although you can force such breaks using the !PAGE directive.

The !WIDTH directive must appear before the first !TOPIC directive.

Example

```
!WIDTH 70
```

MAKEHELP's word-wrap will leave a maximum of 68 characters on each line of help text.

!NOWRAP
!WRAP

!NOWRAP disables the word wrapping that MAKEHELP normally performs. Topic help lines are transferred directly from the input text file to the binary help file when wrapping is off. !WRAP turns wrapping on again. Wrapping is on by default. Use the !NOWRAP directive if you've formatted the help text exactly as you want it before running MAKEHELP.

!SCROLLING
This directive simply sets a flag (in the help file's header) that indicates the help text was optimized for a ScrollingHelpWindow. Routines in OPHELP can detect this flag and use the appropriate object to display the help. Help files are non-scrolling by default. The output file MAKEHELP creates is the same in either case (with the exception of the flag), and both PagedHelpWindow and ScrollingHelpWindow can read either kind of file.

!TOPIC Number [Name]
Each help topic must begin with a !TOPIC directive. Typically, help topics will be entered in ascending numerical order, starting with Number 1. Leaving large gaps in the topic number sequence will waste space in both the help file and in memory. (The amount of space wasted has been significantly reduced in **Object Professional**, however.) Topic number 0 is reserved by the help system.

The !TOPIC directive may also specify an optional topic name, which starts with the first non-blank character following the topic number. The name may contain spaces, and it will later appear on-screen exactly as entered in the !TOPIC directive. (The topic name optionally appears as the title of the help window, and optionally as an entry in the topic index.)

The total space for Names in the help file is limited to 64K bytes. Names are packed one after another in the file, each consuming the actual length of the string plus a length byte. MAKEHELP assures that the 64K limit is not exceeded.

Lines (even blank ones) immediately following the !TOPIC directive start the help text for that topic, which continues until the next !TOPIC directive or the end of the TXT file.

Each help file must have at least one help !TOPIC.

Examples

```
!TOPIC 1 First Help Section
!TOPIC 22
```

Topic 1 has a name, which may appear as the title of the help window when that topic is displayed. The top border of the help window for topic 22 will always be empty, and that topic will never appear in the master help index.

!INDEX IndexNumber
!NOINDEX
These directives control whether a particular topic will appear in the master topic index, and if so, where within the index. The !INDEX and !NOINDEX directives, if used, should appear on the line following each !TOPIC directive. If they are not used for a given topic, MAKEHELP indexes that topic and sorts it into alphabetical position within the master index.

Exclude any named topic from the index by specifying the !NOINDEX directive.

Change the order in which indexed topics appear by using the !INDEX directive followed by an integer IndexNumber. Low values of IndexNumber sort to the beginning of the topic index, and high values to the end, overriding the normal alphabetical order. Specifically, the master topic index is sorted in the following order:

```
Topics with IndexNumber < $7777 in ascending numeric order
Topics without !INDEX directives in ascending alphabetic order
Topics with IndexNumber > $7777 in ascending numeric order
```

As you might guess, MAKEHELP assigns a default IndexNumber of $7777 (30583 decimal) to each topic, and the !INDEX directive overrides this value. When MAKEHELP sorts the topics

for the help index, it uses IndexNumber as a primary key, and the alphabetic topic name as a secondary key.

Do not specify an IndexNumber of $FFFF (65535 decimal) since MAKEHELP uses this special value to disable indexing for a topic.

Examples

```
!TOPIC 1 Help on Help
!INDEX 1
...
!TOPIC 2 A Very Minor Subtopic
!NOINDEX
...
!TOPIC 3 An Alphabetic Approach
...
!TOPIC 4 The Last Word
!INDEX 32000
...
```

We want the "Help on Help" topic to be listed first in the topic index, so we follow its !TOPIC directive with an !INDEX directive specifying item number 1.

Topic 2, describing a very minor subtopic, should not appear on the index at all, so we remove it with the !NOINDEX directive.

Topic 3 should appear in the help index in normal alphabetic order, so it needs no !INDEX directive.

Topic 4 should be the last topic in the help index, so we give it a large IndexNumber.

!LINE

This directive forces MAKEHELP to finish the current line of text, inserting a line break. Since MAKEHELP automatically fills and word-wraps the text in your original help file, the line breaks in the final help screens are not necessarily the same as those in the original text file (unless wrapping is turned off). The !LINE command gives you appropriate control when needed. MAKEHELP also inserts a line break whenever it encounters a blank line or a line that begins with one or more spaces.

!PAGE

This directive forces MAKEHELP to finish the current page of text, inserting a page break. Normally, OPHELP paginates the help text only when the help is displayed. For best appearance, however, you may wish to force page breaks at selected locations in your help file. The ScrollingHelpWindow object treats page breaks no differently than line breaks.

!INCLUDE IncludeFileName

The specified file is read into the help text stream at this point. The include file capability allows for much larger help systems without using a virtual text editor to create them.

MAKEHELP is configured to allow only one level of include file nesting (include files may not themselves use the !INCLUDE directive). Nesting can easily be increased by changing a constant (MaxIncludeNest) in the MAKEHELP.PAS source code.

Example

```
!WIDTH 50
!NOWRAP
!INCLUDE HELP.TX1
!INCLUDE HELP.TX2
```

This might be the complete top level TXT file for a particular help system. All of the topics are contained in the two include files HELP.TX1 and HELP.TX2.

!BIAS BiasAmount

BiasAmount is an integer value between -65000 and +65000. All references to topic numbers following this directive are biased by the specified amount. This directive is useful if you're merging several independent help files into one large one (as would be helpful for POPHELP).

Example

```
!BIAS 500
```

Adds 500 to all topic number references for the text that follows this directive. Note that both cross-references to topics and the topic definitions themselves are adjusted.

You can enter comments in the text file--they won't appear in the final help. Comments are indicated when a semicolon appears in the first column of any line.

```
;ANY TEXT APPEARING AFTER THE SEMICOLON IS IGNORED.
```

Help File Control Characters

Special control characters provide another method to customize the appearance of help displays. With these, you can activate special video attributes; or you can specify that certain passages cross-reference other help topics.

First, we'll describe the characters that control special video attributes. There are three of these:

```
^A  toggles special attribute A   ColorSet.FlexAHelpColor,FlexAHelpMono
^B  toggles special attribute B   ColorSet.FlexBHelpColor,FlexBHelpMono
^C  toggles special attribute C   ColorSet.FlexCHelpColor,FlexCHelpMono
```

The help window's constructor plucks the specified data fields out of a ColorSet variable to use for these special attributes.

Place the control characters anywhere within the help text (not within help directives, however). The characters themselves will not be displayed and do not affect the length of lines as far as word-wrapping is concerned. The first instance of a control character causes the following text to appear in the associated color. The next appearance of the same character toggles that color off, returning the active color to its value prior to the previous control character. Use of these toggles may be nested, up to a maximum of 4 levels. The logic is the same as that of OPCRT's FlexWrite procedure.

To enter control characters using the Turbo Pascal editor, first press <CtrlP>, then hold down the Ctrl key and press the desired character. For example, to enter a ^A into the text, press <CtrlP><CtrlA>.

Two additional control characters activate the cross-reference capability of the help system. A cross-reference is a passage of text that refers to another topic within the help system. While help is displayed, the user can press cursor keys to position a highlight bar (for PagedHelpWindow) or the hardware cursor (for ScrollingHelpWindow) over the cross-reference passage. Then he or she can press <Enter> to select the referenced topic and display its help, and thus browse through many related topics. By default, any help topic can define up to 50 cross-references.

You can use the cross-reference capability to achieve either of two effects. The first is to create local indexes: an entire help topic might be a list of other topic names. In this way, you could have hierarchical indexes, with the master index showing only the most general concepts, and

each of those associated topics containing a local index that pointed to more detailed information. The second effect is what's popularly referred to as "hypertext." In this case, passages of text in one topic are used as links into related topics, hence allowing the user to browse through all the help text in a self-directed manner. ENTRY.TXT demonstrates the hypertext effect; the **Object Professional** help database used with POPHELP shows the local index technique as well.

Control characters within the help text must therefore provide two cues to OPHELP. First, they must indicate which help topic will be activated when a given cross-reference is selected. Second, they must denote the extent of the highlighted passage of text. MAKEHELP uses the following scheme for this purpose:

```
^Dnnnn^Etext^E
```

The control character ^D precedes the referenced topic number "nnnn," which is expressed using decimal digits. The topic number is terminated by ^E, which is immediately followed by the text passage to be highlighted. A second ^E terminates the highlighted text. Note that the ^E characters must be balanced while the ^D appears alone. Here's an example of two cross-references:

```
^D10^ECompiling^E the help text is easy using ^D5^EMAKEHELP^E.
```

In the real help text, the characters shown with carets will be real control characters, of course. If the user positioned the highlight bar over the word MAKEHELP and pressed <Enter>, the help for topic number 5 would appear.

Text marked with ^E is displayed in either of two attributes. When the cross-reference has been selected (in PagedHelpWindow), the text appears in the attribute ColorSet.SelXrefColor (or SelXrefMono). When the cross-reference is not selected, the text appears in the attribute ColorSet.UnselXrefColor (or UnselXrefMono).

Zero has a special meaning when used as the topic number in a cross-reference. After selecting this cross-reference, instead of getting another topic, the user will get the master topic index (and the help window will switch into pick mode). The help window provides the same behavior if the user presses <F1>, but this special cross-reference makes it easier for you to remind people of the master index.

The control character ^F offers a way to embed certain literal characters in the help text that aren't easily obtainable otherwise:

```
^Fnnnn
```

"nnnn" is any sequence of digits (0-9) that specify the ASCII value of the character. You can't use ^F to trick OPHELP into producing on-screen images of the characters ^A..^F, ^L, or ^M, however. OPHELP will still interpret these characters as it usually does. Probably the most valuable way to use the ^F option is to specify

```
^F26
```

which gives the effect of a ^Z character. ^Z terminates text files, so you can't embed a literal one in your help text. ^Z produces a right-arrow character when displayed on-screen, however, and it's sometimes nice to have this appear in help text.

One final note: you can use IBM line-drawing characters (or other characters whose ASCII value exceeds 128) in your help files. OPHELP will display them literally. Most text editors

(Turbo Pascal's included) allow you to enter such characters by holding down the <Alt> key, typing in the ASCII value on the numeric keypad, then releasing the <Alt> key.

Running MAKEHELP

MAKEHELP.EXE is a command-line driven utility that reads a text file you've created according to the rules just described, and writes a binary help file ready for use by your program. You call MAKEHELP from the DOS command line as follows:

```
MAKEHELP [Options] HelpTextFile [HelpBinaryFile]
```

MAKEHELP reads from the text file named by HelpTextFile and creates a new binary file HelpBinaryFile. It applies a default extension of .TXT to HelpTextFile. If HelpBinaryFile is not specified, MAKEHELP uses the text file name after forcing the extension .HLP onto it. MAKEHELP overwrites any existing HelpBinaryFile without warning. The input file and output file must have different names. For example,

```
MAKEHELP DEMO.TXT DEMO.HLP
MAKEHELP DEMO
```

These two commands do the same thing.

MAKEHELP supports the following options:

```
/L   write topic cross-reference listing to HelpTextFile.LST
/Q   don't display warnings (quiet mode)
/S   sort cross-reference listing in index order
```

If your help file has lots of cross-referencing, the /L option may be useful. It shows all the links between topics. Here's a partial example created from ENTRY.TXT.

```
Topic Cross-Reference
Topic    Index  Topic Name
    1  <alpha>  Name field
    2  <alpha>  Address field
                    to:     6
    3  <alpha>  City field
    4  <alpha>  State field
    5  <alpha>  Zip code field
    6  <alpha>  Phones field
                    to:    16    17
                  from:     2    15    24
```

The topics are listed in numerical order. The Index column shows how the master topic index will order the topics: alphabetically (alpha), numerically (the index position shows), or not at all (blanks in this column). The Topic Name column shows the assigned topic name, or <no topic name> if none. If the topic has cross-references, like number 2, the listing shows the topic numbers that it links to. If other topics contain cross-references to this one, like number 6, a "from" line indicates the topics that reference it.

The /S option combined with /L produces the same list, sorted into the order in which the master topic index will appear.

The /Q option disables the display of MAKEHELP warnings (but not fatal errors). MAKEHELP produces only two categories of warnings:

Cross-reference straddles line break
When the text enclosed between ^E characters straddles more than one line, it becomes possible for that same text to be broken across two pages (in a PagedHelpWindow display only). The text on the second page will be highlighted properly, but OPHELP can't select the

referenced topic there. This warning can usually be ignored, since a) it's unlikely that the broken line will also straddle a page break, and b) even if it does, the resulting problem is minor.

Topic cross-references itself
A cross-reference number specifies the same topic where the cross-reference is found. This has no disastrous side effects, but will probably confuse the end-user. Note that MAKEHELP does not detect indirect cross-reference loops, e.g., where topic A references topic B references topic A.

Once MAKEHELP starts running, it makes two passes through the input text: one to count various essential items, such as the number of topics; and a second to format the text and write compressed output.

If MAKEHELP finds fatal errors, it halts and reports the error type and line number within your help text file. MAKEHELP reports errors in the following situations:

Invalid command line option
An option besides /L, /Q, or /S was specified on the DOS command line.

Too many filenames specified
More than two filenames were on the DOS command line.

Input and output filenames must differ
You can't write the MAKEHELP output onto the input file. (You can fool this error check if you try, but that's not advisable!)

Disk I/O problems
Assure that the drives are ready and there is adequate disk space for storage of the output file.

Topic name array exceeds 64K
Use shorter topic names or reduce the number of topics in the help file.

Insufficient memory
MAKEHELP doesn't have enough memory available for temporary storage. Free up some memory by getting rid of TSR's, or reduce the size of the help file.

File index array exceeds 64K
Reduce the largest topic number in the help file. (Each element in the file index array uses 6 bytes.)

Number of cross-references exceeds 16380
There are more cross-references than the /L listing can handle. Don't specify /L and the help file will compile successfully.

Too many nested files
An include file contains a !INCLUDE directive. Remove the nesting or increase the MaxIncludeNest constant and recompile MAKEHELP.

No help topics specified
There were no !TOPIC directives in the input file.

Invalid cross-reference topic number
The number following a ^D character is not valid.

Invalid literal character
The number following a ^F character is not valid.

Invalid width
Invalid item
Invalid index
The number following a help directive is not valid.

Duplicate help topic number
The same index was assigned to more than one help topic.

WIDTH statement must precede first help topic
Move the !WIDTH directive earlier in the help file.

INDEX statement must follow TOPIC statement
Each !INDEX or !NOINDEX directive must follow the !TOPIC directive it refers to.

Unrecognized directive
Column 1 of the help file contains an exclamation point, but the text immediately following it is not a valid help directive.

After successfully processing a help file, MAKEHELP reports some statistics. Here's an example produced with the ENTRY.TXT input file:

```
Summary ..........
   525 total lines in help file
    24 topic slots used                          HighestTopic
    24 indexed topics                            PickTopics
    31 cross-references
    48 bytes in topic name index                 SizeOf(Topic Name Index)
   398 bytes in packed topic name table          SizeOfNames
    48 bytes in sorted topic map                 SizeOf(Master Topic Index)
   144 bytes in help index                       SizeOf(Disk Index)
  3682 bytes in largest topic (topic 15)         BiggestTopic
    62 bytes in longest line of help
     5 most cross-refs in a topic (topic 13)
 17562 uncompressed help bytes
 12537 compressed help bytes
  0.71 compression factor
```

The names to the right of the listing match those used in the discussion "Organization of a Help File" below. Many of these items are self-explanatory (or will be after you read the following sections). A few notes on those that aren't: "Topic slots" is the same as the largest topic number. Note that it counts unused topic numbers. "Indexed topics" counts the number of named topics for which the !NOINDEX directive wasn't specified. The "packed topic name table" size is the length of all the topic name strings, including the length byte of each one. The largest topic size is in uncompressed bytes and includes space for "decompression overlap" (see below). The length of the longest line (after wrapping) may aid you in deciding how wide your help window should be. "Most cross-references in a topic" must be smaller than the OPHELP constant MaxXrefsPerSection, or else some cross-references will be ignored. Compressed help bytes shows how many bytes of the help file are actually used for help text as opposed to indexing and topic name information.

Binding Help into a Program

OPHELP can directly use the binary file produced by MAKEHELP, without further processing. However, if the help file is relatively small (definitely less than 64K), you may want to link it into your application so that no additional files are required at runtime.

You can do this with the aid of the BINOBJ.EXE utility that Borland supplies with Turbo Pascal. BINOBJ converts the binary help file into a linkable OBJ file that can be bound into your program.

Let's use an example to show how this works. Assume you have a help text file named MYHELP.TXT. The first step would be to run MAKEHELP to create the binary help file:

```
MAKEHELP MYHELP.TXT MYHELP.HLP
```

Then run BINOBJ to create the OBJ file:

```
BINOBJ MYHELP.HLP MYHELP.OBJ MYHELP
```

The third parameter to BINOBJ specifies the name of the public identifier by which your program can refer to the contents of the OBJ file.

Now, within your application program, insert the following lines of source code:

```
{$L MYHELP.OBJ}
procedure MyHelp; external;
```

The $L compiler directive tells Turbo to link in MYHELP.OBJ. The fake procedure MyHelp gives your program a way to refer to the contents of that OBJ file.

Finally, to initialize the help system, you can call one of the constructors InitMem, InitMemCustom, or InitMemDeluxe. For example,

```
if not PH.InitMem(X1, Y1, X2, Y2, @MyHelp, PickVertical) then
  {error initializing};
```

The notation "@MyHelp" passes the address of the linked-in help file to the constructor, which validates that properly formatted help data is at that location.

From then on, use the help object's methods just the same as if a disk file contained the help.

Using OPHELP with CommandWindows and CommandProcessors

Each CommandWindow derivative (PickList, DirList, Memo, TextEditor, Menu, etc.) is associated with a CommandProcessor object. And each CommandProcessor supports a context-sensitive help hook. If the hook is used, the keystrokes associated with the CommandProcessor's ccHelp command will cause the CommandWindow's Process method to call a user-written procedure that displays context-sensitive help. The techniques for activating the hook are described in the OPCMD chapter.

The best action that the user-written procedure can take is to call upon OPHELP to display a help topic specific to the context. This works very naturally if you do some advance planning. When used in combination with OPHELP, the user-written procedure takes the form:

```
{$F+}
procedure MyGetHelpProc(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
  if HelpIndex <> 0 then begin
    MyHelpWindow.SetTopic(HelpIndex);
    MyHelpWindow.Process;
    MyHelpWindow.Erase;
  end;
end;
```

This simple routine counts on two factors: first, that the help window (MyHelpWindow) has been initialized earlier in the program; and second, that each window object has been assigned a range of unique help topic numbers for which the help file contains information. In this way, the HelpIndex passed into the user routine is sufficient to determine exactly what help topic to display.

Some window and editor types–Memo, Browser, LineEditor, and others–will require only a single help topic to describe them. You assign this topic number by calling the SetHelpIndex method of the CommandWindow or LineEditor type. Note that each *instance* of an object can have a unique help topic if desired. In this way, different line editors could have different help information or they could share standard help about edit keystrokes, etc.

Other window types–PickList, Menu, EntryScreen, and others–are more likely to require multiple topics for each instance of the object. For example, a Menu may require a topic for each menu item. Each of these objects provides a way to specify unique topic numbers for every item if you desire.

If you need a unique help topic for each item in a PickList, extra effort will be required to write the help hook routine. The reason is that PickList assigns a topic number equal to the item number, which always ranges from 1 to the number of items. If you have only one PickList, then you can reserve help topics starting at 1 for the list. If you have more than one, you must come up with a biasing scheme: PickListA's topics are in the range PickABase+1.. PickABase+NumberInA, PickListB's are in PickBBase+1..PickBBase+NumberInB, and so on. This is easy enough to do, but requires that you have global access to the pick variables or some other global flag. Assuming you do, the help routine looks something like:

```
{$F+}
procedure MyGetHelpProc(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
  if HelpIndex <> 0 then begin
    if UnitCode = ucPick then begin
      {Help request came from OPPICK}
      if IdPtr = @PickListA then
        inc(HelpIndex, PickABase)
      else if IdPtr = @PickListB then
        inc(HelpIndex, PickBBase);
      {else take the HelpIndex passed in}
    end;
    MyHelpWindow.SetTopic(HelpIndex);
    MyHelpWindow.Process;
    MyHelpWindow.Erase;
  end;
end;
```

Remember that the object requesting help passes a UnitCode that indicates which unit the request came from, and an IdPtr that generally equals the address of the object instance. In this

example, we can bias the HelpIndex number by comparing IdPtr to each of the possible PickLists that might request help.

Of course, there are other ways to associate context-sensitive help with the objects of a program, but they all come down to mapping topic numbers in a logical manner.

1. AbstractHelpWindow: Help File Access

AbstractHelpWindow is an abstract object: you shouldn't declare variables of this type. However, it does implement a lot of functionality for its children, PagedHelpWindow and ScrollingHelpWindow. In fact, the two children have only a few non-internal methods of their own, mostly constructors specific to their requirements. As a result, many of the methods that you'll use for help systems will be described in this section, so that we don't need to repeat them for each type of help window. We'll defer complete example programs to those later sections; fragmentary examples shown for each method apply to either derived object except as stated.

As you think about an AbstractHelpWindow keep in mind that it is derived from the PickList object. What it inherits is the ability to display and select from the master topic index. What the AbstractHelpWindow adds is the ability to access a help file and load a specified topic into a memory buffer. PagedHelpWindow and ScrollingHelpWindow then add the code to display the information in this buffer and to accept user commands to browse around and select cross-references to other topics.

Note again that the help window types are "dual-mode" windows. If you call the Process method of a newly constructed help window, it will act just like a PickList to display the master topic index; this is called pick mode. If you use SetTopic to specify a valid topic number before calling Process, it will display the help text; this is called help mode. If the user executes the ccIndex command (mapped to <F1> by default) while in help mode, the window automatically switches back to pick mode.

Organization of a Help File

It isn't essential for you to know how a help file is organized, but it may aid in optimizing and understanding your help system. As we've mentioned before, the same file format is used for all OPHELP help window types. AbstractHelpWindow provides the methods to access the file, read the compressed help text into a buffer, and decompress it.

The help file created by MAKEHELP is divided into the following sections:

```
Section            Size
-------------------------------------------------
HelpHeader         SizeOf(HelpHeader)
Topic Name Index   HighestTopic*SizeOf(Word)
Topic Name Buffer  SizeOfNames
Master Topic Index PickTopics*SizeOf(Word)
Disk Index         HighestTopic*SizeOf(HelpIndexRec)
Help Topics        varies
```

The file begins with a fixed-length header (34 bytes). This header includes a signature ('OPH1') used to recognize a valid help file, and various other information about the number of topics and the length of the various sections that follow. One important field is HighestTopic, which is the largest topic number found in the original help text file. (See type HelpHeader below for more details.) OPHELP reads the header when it opens a new help file in order to validate it and to initialize various in-memory structures of the help object.

Next is the Topic Name Index, which is an array of Word ranging from 1 to the highest topic number. Each element in the array tells OPHELP where to find the topic's name string, if any, in the Topic Name Buffer that follows. For any topic that is unused or has no topic name, the Topic Name Index holds $FFFF; otherwise it holds the name's offset (starting from 0) in the Name Buffer.

Topic names are stored in the Name Buffer in standard Turbo Pascal string format--a length byte followed by the unencrypted, uncompressed string. Strings are packed immediately after one another so that no space is wasted in this array. The array uses SizeOfNames bytes, where SizeOfNames is another field stored in the help header and reported by MAKEHELP. OPHELP always accesses the Name Buffer indirectly by using the Topic Name Index. These names are used to provide the optional topic titles on the help window and the item strings for the master topic index's pick list.

The Master Topic Index is a Word array that maps a pick list item number (1..PickTopics) to the associated topic number. This array filters out unused and unindexed topics, and accounts for the alphabetic and numeric sorting of the topic index.

The Disk Index tells OPHELP where it can find the text for each help topic. Each element in this array is a 6-byte record which holds a LongInt seek offset to the beginning of the topic and a Word value that specifies the number of bytes of compressed text there. The seek offset is $FFFFFFFF for non-existent topics.

You can see from the file's organization that 8 bytes of space are wasted for each unused topic number in the range 1..HighestTopic. That's not enough to worry about for small gaps in the topic number sequence, but it does indicate that you'll want to avoid a three topic help file with topics numbered 1000, 5000, and 10000. (This would result in almost 80,000 wasted bytes of file space.)

Following the Disk Index is the text for each topic. The data is compressed using a fast, simple method similar to that in Borland's Word Wizard Toolbox. In brief: the 15 most frequently used characters in the entire help file are each replaced with a 4-bit code in the range 0..14. The remaining 240 characters are encoded with 12 bits each: the 4-bit value 15 followed by the 8-bit character itself. The encoding table is stored in the help header. This algorithm typically results in 30% overall compression. Compression, of course, is performed by MAKEHELP. OPHELP decompresses each topic when it is requested.

OPHELP allocates heap space to buffer some of these structures in memory. It accesses the help header so frequently that it immediately copies this small record onto the heap as soon as the help file is opened. Whenever the help window is used in pick mode, OPHELP temporarily allocates buffers to hold the Topic Name Index, Topic Name Buffer, and Master Topic Index. If there isn't sufficient heap space for these structures, you can't use pick mode. These structures must be kept in memory for the PickList to provide adequate performance.

If the hwStaticNameBuffer option is enabled, OPHELP allocates and initializes these three buffers as soon as the help file is opened, and leaves them allocated until the window is disposed. This option saves the time of reading these structures whenever the master topic index is requested. It's an especially good idea for TSR's that use OPHELP (POPHELP, for example); since the heap space must be reserved anyway, why not keep it allocated?

OPHELP allocates one additional heap block, the decompression buffer, which is large enough to hold the largest topic in the help file after it is decompressed. It reads each compressed topic into the top of this buffer, then decompresses the text into the bottom of the buffer. The difference between the actual buffer size and the largest topic size is called the "decompression overlap"; this extra space keeps the plain text from overwriting the compressed text during decompression. MAKEHELP reports the actual block size, BiggestTopic, in its summary.

When the help file is linked into an application, OPHELP allocates only the decompression buffer on the heap. It simply stores pointers to the other "file" sections, which are already in memory. In this case, the hwStaticNameBuffer option has no effect.

The Previous Topic Stack

When people take advantage of the cross-referencing capabilities of a hypertext-like help system, they often want to be able to back up to a topic that was previously displayed. AbstractHelpWindow supports this with a data structure called the previous topic stack. This structure is implemented using OPROOT's StaticQueue object which, in spite of its name, works well for certain kinds of stacks.

When you call a help window constructor, it allocates a StaticQueue large enough to hold MaxPreviousTopics, a constant that defaults to 20. The heap space used by this StaticQueue is different for a PagedHelpWindow and a ScrollingHelpWindow, since they must store different amounts of information to return to a previous topic.

Each time the user selects a new help topic, the previous topic's state (the topic number, its page number or scroll position, and its current cross-reference link or cursor position) is pushed onto the previous topic stack. If the user enters the ccPrevTopic command (mapped to <AltF1> by default), the most recent topic is popped off the stack and the help window is updated to show that information. If the stack is empty, ccPrevTopic is ignored. If the stack is full, the oldest topic is lost when a new one is pushed.

By default, OPHELP retains the contents of the topic stack between one call to Process and the next. You can force OPHELP to clear the stack each time Process exits by setting the hwClearStackAtExit option.

Mouse Support

Mouse support is built into the help window objects; all you need to do is activate it. To try it, add the following lines to the introductory HelpExample, following the call to PH.InitCustom:

```
if MouseInstalled then begin
  with DefaultColorSet do
    SoftMouseCursor($0000, (ColorMono(MouseColor, MouseMono) shl 8)+$04);
  HelpCommands.cpOptionsOn(cpEnableMouse);
  PH.hwFrame.AddHotSpot(frTL, #4, DefaultColorSet);
  PH.SetPrevTopicHotSpot(hsSpot, frTL);
end;
```

This code first assures that a mouse is installed. If so, it defines the mouse cursor characteristics. This step is optional since a software mouse cursor is the default, but you'll probably want to set its colors to be consistent with the rest of your ColorSet. You should always use a software (as opposed to hardware) mouse cursor with the help window objects. See OPMOUSE for more information about defining the mouse cursor.

The call to HelpCommands.cpOptionsOn initializes the help window's command processor to support the mouse. See OPCMD for more information, especially if you are using background task hooks.

Finally this code adds and activates a special purpose hot spot for the help window. The hot spot in this case is a character #4 (a diamond) located in the upper left corner of the frame. The hot spot is added to hwFrame so that it appears only when the window is in help mode. SetPrevTopicHotSpot maps this hot spot to the help window's previous topic command. In this way, the help window's Process method knows to pop the topic stack whenever the user clicks on the hot spot at the upper left corner of the window.

The mouse also works with scroll bars in the help window. We already added vertical scroll bars in our example, so we don't need to add them here. As usual, the minimum (0) and maximum (MaxLongInt) user limits are arbitrary when the scroll bar is added; the help window updates

them when it knows more about the help topic. The PagedHelpWindow supports only vertical scroll bars while in help mode; horizontal scroll is unneeded. ScrollingHelpWindow supports both horizontal and vertical scroll bars. The sliders of both scroll bars change as you page or scroll through the help information. If a ScrollingHelpWindow's hwMousePage option is enabled, clicking on the increment/decrement arrows of the vertical scroll bar brings a new page of help into view; otherwise, the help scrolls line by line.

You can select topic cross-references using the mouse with either type of help window. The first click over a cross-reference moves the highlight bar or the hardware cursor to the mouse position. The second click in the same location selects the referenced topic just like pressing <Enter>.

Pressing both mouse buttons while the window is in help mode switches it to pick mode, displaying the master topic index.

Specialized Frame Headers

Two specialized header types provide visual enhancement of help windows when operating in help mode. (In pick mode, the specialized headers applicable to the PickList are available as usual.)

The first new header is just like PickList's "more header," but it shows whether there is more help text outside of the window. It may be used instead of a scroll bar. Add a "more help header" using the following method:

```
procedure AddMoreHelpHeader(PromptStr : String80;
                            Posn : HeaderPosType;
                            UpStr, DnStr, SepStr : String10;
                            UpPos, DnPos, SepPos : Byte);
  {-Add a specialized more header to indicate more help}
```

Instead of adding the header to the window's main frame (wFrame), AddMoreHelpHeader adds it to the special help mode frame, hwFrame. The introductory HelpExample program shows how to do it:

```
PH.AddMoreHelpHeader(' PgUp/PgDn for more ', heBR,
                     'PgUp', 'PgDn', '/', 2, 7, 6);
```

The parameters here are interpreted just like those of AddMoreHeader in PickList.

The second new header is a topic name header. Use this header if you want each topic's name to appear when that topic is displayed. Add a topic name header with the following method:

```
procedure AddTopicHeader(PadSize : Byte; MaxLen : Byte; Posn : HeaderPosType);
  {-Add a header to display the topic name}
```

PadSize is the number of spaces to surround the topic name; MaxLen is the longest length to accept (longer names are clipped); and Posn specifies the frame location for the header. Again, HelpExample shows how:

```
PH.AddTopicHeader(1, 60, heTC);
```

In this case, the header is centered in the top border, each topic name is limited to 60 characters and padded with one space on each side.

The User Move Hook

Help windows inherit the user move hook from the PickList. See the discussion there for background. When applied to a help window, the move procedure must detect the current

mode of the window in order to provide accurate information. AbstractHelpWindow's InHelpMode method is designed for this purpose:

```
function InHelpMode : Boolean;
  {-Return True if window shows help, not pick list}
```

An appropriate move routine would be written like this:

```
{$F+}
procedure MyMoveProc(P : PickListPtr);
begin
  if AbstractHelpWindowPtr(P)^.InHelpMode then
    {perform actions appropriate to help mode}
  else
    {perform actions appropriate to pick mode};
end;
```

Note the typecast required to inform the compiler that the pointer refers to the address of a AbstractHelpWindow instead of just a PickList. Additional typecasting may be necessary if the move procedure refers to a PagedHelpWindow or ScrollingHelpWindow.

The Nested Function Hook

The nested function hook is a routine that the Process method calls whenever it detects the ccNested command. By default, ccNested is not mapped to any keystrokes. A nested function must be compiled under the FAR model (with the {$F+} compiler directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
function MyNestedFunc(HPtr : AbstractHelpWindowPtr) : Boolean;
```

OPHELP interfaces a default nested function (NoNestedAction) that does nothing but return False.

Activate your own nested function by assigning a keystroke to ccNested (see CommandProcessor.AddCommand) and by calling the help window's SetNestedFunc method, passing your function as a parameter:

```
procedure SetNestedFunc(NF : hwNestedFunc);
  {-Set function to call for nested xref commands}
```

The pointer passed to the nested function contains the address of the current help window. If the nested function returns True, the Process method will exit, returning control to your program. Unless the nested function changes the value of the stored command code, the exit command will be ccNested. If the function returns False, Process will loop around to get more commands.

What's the purpose of this function? To tell the truth, we've never used it in more than a test program, but the idea is this: the nested function could detect which cross-reference, if any, was currently highlighted. It could then pop up a prompt window to get further information from the user. The original suggestion that lead to this feature was for a "smart form": the help text would provide instructions on how to fill out the form, and the cross-reference links would serve as icons for entry of data.

Taking advantage of the nested function will require somewhat advanced knowledge of the help window objects. If you implement something nifty with it, we'd be curious to hear about it.

Constants

```
hwStaticNameBuffer = $0001; {Preallocate and keep name buffers at all times}
hwPickIndex        = $0002; {Enable ccIndex command}
hwClearStackAtExit = $0004; {Clear the previous topic stack when exiting help}
hwMousePage        = $0008; {Clicking on scroll bar scrolls by page}
hwAdjustScroll     = $0010; {Adjust horizontal scroll limits automatically}
hwBlockMark        = $0020; {Enable block marking}
```

These are the options that affect the behavior of AbstractHelpWindow, PagedHelpWindow, and ScrollingHelpWindow. hwStaticNameBuffer forces each help window constructor to allocate heap space for the topic name index, the topic name buffer, and the master topic index (see "Organization of a Help File" above) and to load these buffers immediately. These heap buffers remain allocated until the window is disposed. If hwStaticNameBuffer is not enabled, the help windows allocate and load these buffers only when the window needs to display or search the master topic index, and deallocate them as soon as the operation is finished.

hwPickIndex enables the ccIndex command. If hwPickIndex is disabled, the user will not be able to access the master topic index from within a help window.

If hwClearStackAtExit is on, the Process method clears the previous topic stack whenever it exits.

hwMousePage affects only the ScrollingHelpWindow. If enabled, a mouse click on the increment/decrement arrow of a vertical scroll bar causes the window to scroll a full window height; otherwise, it scrolls a single line.

hwAdjustScroll also affects only the ScrollingHelpWindow. If enabled, the help window constructor automatically sets the horizontal scrolling limit of the window to match the longest line in the help file (stored by MAKEHELP). If disabled, the window allows essentially unlimited horizontal scrolling unless a limit is set by calling SetMaxScroll.

hwBlockMark determines whether block marking commands are available in the ScrollingHelpWindow.

```
DefHelpOptions : Word = hwPickIndex+hwAdjustScroll;
```

The default options for a help window.

```
BadHelpOptions : Word = hwModeChangePending+hwProcessingHelp+
                        hwHelpFrameInit+hwHelpModeActive+hwHelpFileOpen+
                        hwHelpInMem+hwNamesLoaded;
```

Options used internally by help window. The hwOptionsOn and hwOptionsOff methods will not change these bits of a help window option word.

```
HelpKeyMax = 196;
HelpKeyID  : string[16] = 'ophelp key array';
HelpKeySet : array[0..HelpKeyMax] of Byte = (...);
HelpCfgEnd : Byte = 0;
```

HelpKeyId is an ID string used to mark the beginning of the configuration data area for OPHELP; HelpCfgEnd marks the end of the configuration data area. In between them is HelpKeySet, the command table used by HelpCommands. HelpKeyMax is the last valid index into the table.

```
HelpFileOpenMode : Byte = $20; {FileMode: read only, deny write}
```

The FileMode used to open a help file. Help window constructors use this value when they are called. If the DOS version is less than 3.0, they override this value and set it to 0, read-only, since earlier DOS versions didn't support the extended modes.

```
MaxPagesPerSection  = 20;
```

Maximum number of help pages per topic. This limit applies only to the PagedHelpWindow object. A page break occurs when the number of lines exceeds the window height (at the time the topic was loaded) or an explicit page break is encountered (resulting from a !PAGE directive in the original help text). Pages in excess of MaxPagesPerSection are ignored.

```
MaxXrefsPerSection  = 50;
```

Maximum number of cross-references per topic. This limit applies to both PagedHelpWindow and ScrollingHelpWindow. Cross-references in excess of MaxXrefsPerSection are drawn in the special video attribute for unselected links, but they cannot be selected.

```
MaxPreviousTopics   = 20;
```

Capacity of the previous topic stack. When the stack is full and a new topic is pushed, the oldest topic on the stack is "pushed off the end" and lost.

```
FlexStackSize       = 4;
```

Maximum number of nested attributes. Help highlighting activated by the ^A, ^B, and ^C characters may be nested. For example, the string

```
^Anext word ^Bis drawn in ^Canother ^Cvideo ^Battribute ^Aaltogether.
```

would show the word 'another' in the ^C attribute, 'video' in ^B, 'attribute' in ^A, and 'altogether' in the normal text attribute. This example uses three levels of nesting.

```
Attr1Toggle   = ^A;          {Character toggles special attribute 1}
Attr2Toggle   = ^B;          {Character toggles special attribute 2}
Attr3Toggle   = ^C;          {Character toggles special attribute 3}
IndexMarker   = ^D;          {Character marks topic number that follows}
XrefToggle    = ^E;          {Character toggles xref highlight}
LiteralMarker = ^F;          {Character marks literal ASCII char following}
LineBrkMark   = ^M;          {Denotes end of line of help}
PageBrkMark   = ^L;          {Denotes end of page of help}
SectEndMark   = #0;          {Denotes end of help section}
```

OPHELP interprets these characters to format and display your help text. Their meanings are described in a previous section.

```
HelpId : array[0..3] of Char = 'OPH1';
```

Valid compiled help files start with this four character sequence.

```
NoHelpAvailable = $FFFFFFFF;
```

Seek offset (in a HelpIndexRec) that indicates no help is available for a particular topic number.

```
NoTopicName     = $FFFF;
```

Name offset (in a topic name index) that indicates no topic name is available for a particular topic number.

```
XrefForIndex    = 0;
```

If a cross-reference link specifies this topic number, OPHELP will display the master topic index when that cross-reference is selected.

```
PickTopic       = 0;
```

If this topic number is specified when calling SetTopic, the next call to Draw or Process will display the master topic index.

```
HelpExitCommands : ByteSet = [ccPrevTopic];
```

Dynamic exit commands that cause the Process method of the master topic index PickList to exit.

```
WordChars : string[127] = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ_0123456789';
```

The uppercase characters considered to form a word for the purpose of ScreenWordXY parsing.

Types

```
XlateArray = array[0..15] of Byte;
```

The code table used for compressing and decompressing help text. Element 0 is the character most frequently found in a particular help file. Element 1 is the next most frequent, and so on. Element 15 isn't used.

```
HelpHeaderPtr = ^HelpHeader;
HelpHeader =
  record
    ID           : LongInt;    {Marks help file version}
    HighestTopic : Word;       {Highest topic number}
    BiggestTopic : Word;       {Size of largest topic in uncompressed bytes}
    PickTopics   : Word;       {Number of topics in pick index}
    SizeOfNames  : Word;       {Bytes in all topic names (packed table)}
    LongestName  : Byte;       {Longest topic name}
    LongestPick  : Byte;       {Longest name in pick index}
    WindowWidth  : Byte;       {Preferred width of help window}
    Scrolling    : Boolean;    {True if scrolling display is preferred}
    MaxLineLen   : Integer;    {Maximum line length}
    XlateTable   : XlateArray; {Table for decompression}
  end;
```

The header found at the beginning of a help file, and used frequently to manage a help window. Help window methods typically provide the access you'll need to the fields of this record.

```
BytePtr      = ^Byte;
CharArray    = array[0..65520] of Char;
CharArrayPtr = ^CharArray;
WordArray    = array[1..(65521 div SizeOf(Word))] of Word;
WordArrayPtr = ^WordArray;
FilePtr      = ^File;
```

Miscellaneous data types used to manage various character buffers and topic number maps. When CharArray and WordArray types are allocated on the heap, they consume the exact size needed and not the maximum sizes described by these type declarations.

```
HelpIndexRec =
  record
    Start   : LongInt;     {File position of topic}
    CompLen : Word;        {Compressed length of topic}
  end;
HelpIndex    = array[1..(65521 div SizeOf(HelpIndexRec))] of HelpIndexRec;
HelpIndexPtr = ^HelpIndex;
```

Data types for managing the help file's disk index, which specifies the file location and compressed data length of each topic's text. The complete disk index is not read into memory unless the whole help file has been linked into an application.

```
HelpFileSection = (hwHeader,     {Fixed size header}
                   hwNameIndex,  {Offset index for packed topic names}
                   hwNames,      {Packed topic names}
                   hwTopicIndex, {Sorted topic index for pick list}
                   hwFileIndex); {Index to help topics themselves}
```

An enumerated type naming the components of a help file.

```
hwMatchFunc     = function (VaryingName, FixedName : String;
                            HPtr : AbstractHelpWindowPtr) : Boolean;
```

Template for a user-written function that helps to search for a topic by topic name. See the method FindTopic for more information.

```
hwNestedFunc    = function (HPtr : AbstractHelpWindowPtr) : Boolean;
```

Procedure type for the "nested function hook." See the discussion above for more information.

```
AbstractHelpWindowPtr = ^AbstractHelpWindow;
AbstractHelpWindow =
  object(PickList)
    ...
  end;
```

The abstract help window object.

```
Xrefs = 0..MaxXrefsPerSection;
```

Defines the valid range of cross-reference indexes.

```
FlexStack = array[0..FlexStackSize] of Char;
```

Defines the stack of attribute codes used for flexwriting the help text. Note that the stack holds the control characters (^A, ^B, etc.) rather than the video attributes.

Variables

```
HelpCommands : CommandProcessor;
```

The default CommandProcessor for a help window. This variable is initialized by OPHELP's initialization block. See OPHELP.ICD for details of the key assignments.

AddMoreHelpHeader / AddTopicHeader

Declaration

```
procedure AddMoreHelpHeader(PromptStr : String80;
                            Posn : HeaderPosType;
                            UpStr, DnStr, SepStr : String10;
                            UpPos, DnPos, SepPos : Byte);
```

Purpose

Add a specialized more header to indicate more help.

Description

Works just like PickList's AddMoreHeader, except that the header is added to the help window's hwFrame object rather than wFrame and the header contents are adjusted depending on the help text rather than pick list contents.

Example

```
PH.AddMoreHelpHeader(
  ' PgUp/PgDn for more ', heBR, 'PgUp', 'PgDn', '/', 2, 7, 6);
```

Adds a more header to the bottom-right of the help window. This will display 'PgUp' and 'PgDn' strings whenever appropriate. A '/' will be drawn as a separator whenever both 'PgUp' and 'PgDn' are visible.

See Also

InitHelpFrame PickList.AddMoreHeader

Declaration

```
procedure AddTopicHeader(PadSize : Byte; MaxLen : Byte; Posn : HeaderPosType);
```

Purpose

Add a header to display the topic name.

Description

OPHELP modifies this header whenever the help system loads a new topic. If the topic has no name, no header will be visible. Otherwise, the topic name is clipped to MaxLen-2*PadSize characters, if necessary, and padded with PadSize spaces on both ends. The header is displayed at the position indicated by Posn (heTL, heTC, heTR, heBL, heBC, or heBR).

You can have at most one topic header per help window.

If there is insufficient memory, AddTopicHeader will generate the error epFatal+ecOutOfMemory.

Example

```
PH.AddTopicHeader(1, 60, heTC);
```

Adds a header to display the current topic name centered on the top border. The heading will have one blank character on both sides; the maximum width of the heading will be 60 characters.

Declaration
```
function CurrentTopic : Word;
```

Purpose
Return the topic number currently loaded into the help buffers.

Description
This method is useful within user-written premove or nested function routines. It returns the topic number of the last topic loaded and decompressed by the help window. The value returned is the same whether the window is currently in help mode or pick mode.

Example
See SetNestedFunc for an example.

See Also
SetTopic

Declaration
```
function CurrentXrefTopic : Word; virtual;
```

Purpose
Return help topic of current cross reference, $FFFF if none.

Description
This function is useful within premove or nested function routines. If the current help window has cross-reference links and the cursor is over one of them, CurrentXrefTopic returns the topic that the link points to. Otherwise, it returns $FFFF. Note that when CurrentXrefTopic returns 0, the selection of that cross-reference will activate the master topic index.

See Also
SetNestedFunc

Declaration
```
destructor Done; virtual;
```

Purpose
Close and dispose of help window.

Description
The help file, if any, is closed. All heap buffers allocated by the help window are deallocated.

Example
```
PH.Done;
```
Deallocates all heap space used by PH.

See Also
PickList.Done

Draw

Declaration

```
procedure Draw; virtual;
```

Purpose

Choose appropriate frame and draw help window or pick index.

Description

If the window's mode has changed, Draw swaps the hwFrame and wFrame objects. The appropriate frame is drawn and the window's contents are updated (by UpdateContents) to display either the master topic index or the current topic's help.

Draw will not load a help topic by itself. Hence, the following sequence will not generate the results you expect:

```
PH.SetTopic(100);
PH.Draw;              {incorrect!}
```

The only way to load new topics is by calling Process. The proper sequence to display topic 100 is therefore

```
PH.SetTopic(100);
PH.Process;
```

If you want to display a help topic without waiting for keyboard input, you can use ProcessAutoPilot:

```
const
  QuitRightAway : Byte = ccQuit;
...
PH.SetTopic(100);
PH.ProcessAutoPilot(QuitRightAway, 1);
```

If the window is in pick mode when you call Draw, heap buffers for the topic index may need to be allocated and loaded. Hence, it's important to check for errors (by calling GetLastError) after Draw.

Draw is used primarily for internal reasons in help objects. It's documented here to point out these restrictions.

See Also

Process UpdateContents

Declaration

```
function FindTopic(Name : String; MatchFunc : hwMatchFunc) : Word; virtual;
```

Purpose

Return Topic number of help with specified Name, 0 if not found.

Description

FindTopic scans all named topics from 1 to the highest topic, looking for one that matches Name using the specified match function. If a matching topic is found, FindTopic returns its number; otherwise it returns 0.

MatchFunc is a function that has been compiled under the FAR model (using the {$F+} directive), is global, and matches the following prototype definition:

```
{$F+}
function TopicMatchFunc(VaryingName, FixedName : String;
                        HPtr : AbstractHelpWindowPtr) : Boolean;
```

MatchFunc's purpose is to compare VaryingName to FixedName and return True if they are considered to match. The function can use any rules that it wants: it can be case-sensitive or not; it can require an exact or partial match; it can use "sounds-like" techniques; and so on. The FixedName parameter is always the same as the Name string passed to FindTopic. VaryingName takes on each topic's name in turn. HPtr is the address of the help window instance that called it.

OPHELP provides a default match function, DefaultTopicMatch. This routine checks for exact equality after raising both strings to uppercase.

If the hwStaticNameBuffer option has not been enabled, FindTopic must allocate heap space to load the three name arrays described earlier. If it is unable to do so, or if an I/O error occurs, it generates an error whose value can be obtained by calling GetLastError.

Examples

```
PH.SetTopic(FindTopic('notes field', DefaultTopicMatch));
```

Uses OPHELP's built-in string comparison function (DefaultTopicMatch) to find a topic whose name exactly matches 'notes field' when case is ignored.

```
uses
  OpCrt, OpRoot, OpString, OpCmd, OpFrame, OpWindow, OpPick, OpHelp;
var
  PH : PagedHelpWindow;
  {$F+}
  function MyTopicMatch(VaryingName, FixedName : String;
                        HPtr : AbstractHelpWindowPtr) : Boolean;
  begin
    MyTopicMatch :=
      (CompUCstring(Copy(VaryingName, 1, 5), Copy(FixedName, 1, 5)) = Equal);
  end;
  {$F-}
begin
  PH.InitCustom(5, 8, 75, 18, DefaultColorSet, DefWindowOptions or wBordered,
                'ENTRY.HLP', PickVertical);
  PH.AddTopicHeader(1, 60, heTC);
  PH.SetTopic(FindTopic('notes', MyTopicMatch));
  PH.Process;
  PH.Done;
end.
```

Demonstrates a custom matching function. MyTopicMatch returns True for the first topic whose first five characters match the passed string (case-insensitive).

See Also

ScreenWordXY

GetTopicChoice / HighestTopic

Declaration

```
function GetTopicChoice : Word;
```

Purpose

Return topic chosen by pick.

Description

Call this routine after calling Process while in pick mode. It returns the help topic number for which the user requested help. The variable that GetTopicChoice accesses is initialized as long as the user did not exit the pick list with the ccQuit or ccError commands.

Note that PickList.GetLastChoice does not return the help topic number. It returns an unmapped index that is not directly usable.

Example

```
PH.SetTopic(PickTopic);
PH.Process;
case PH.GetLastCommand of
  ccError,
  ccQuit :
    {no selection};
  ccSelect :
    begin
      PH.SetTopic(PH.GetTopicChoice);
      PH.Process;
    end;
end;
```

The first call to Process displays the master topic index and gets a selection. The second SetTopic sets up to display the help topic just selected.

See Also

Process

Declaration

```
function HighestTopic : Word;
```

Purpose

Return highest topic number in help system.

Description

This function accesses a number originally stored in the help file header. If you call SetTopic with a topic number higher than HighestTopic, an error will be generated.

Example

```
if PH.HighestTopic > 100 then
  {take some action};
```

This example takes specific action if the highest topic number is greater than 100. Perhaps this indicates that the user has loaded an alternate help set.

See Also

PickTopics

Declaration
```
function hwOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This function tests the help options, which are described in the Constant declarations near the beginning of this section.

Example
```
if PH.hwOptionsAreOn(hwMousePage) then
  {take some action};
```
Take some action if mouse scrolling is enabled.

Declaration
```
procedure hwOptionsOff(OptionFlags : Word);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadHelpOptions (see the Constants section, above).

If the hwStaticNameBuffer option is specified, hwOptionsOn immediately deallocates the topic name buffers.

Example
```
SH.hwOptionsOff(hwMousePage);
```
Disable mouse scrolling by page.

Declaration
```
procedure hwOptionsOn(OptionFlags : Word);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadHelpOptions (see the Constants section, above).

If the hwStaticNameBuffer option is specified, hwOptionsOn immediately allocates and loads the topic name buffers.

Example
```
SH.hwOptionsOn(hwMousePage+hwStaticNameBuffer);
```
Enable mouse scrolling by page and static name buffers.

InHelpMode / Init

Declaration

```
function InHelpMode : Boolean;
```

Purpose

Return True if window shows help, not pick list.

Description

This function tests an internal flag that indicates whether the help window last displayed help text or the master topic index. A newly initialized help window returns False.

This function is particularly useful in two circumstances: interpreting the ccQuit command after a Process call (see the introductory example); and deciding what a user move routine should do (see "The User Move Hook," above).

Example

```
if PH.InHelpMode then
  PH.SetTopic(PickTopic);
```

If Process exited while displaying help text, switch back to the pick list the next time Process is called. See the Process loop in the introductory program for a similar use of InHelpMode.

See Also

Process SetTopic

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte;
                 HelpFile : PathStr;
                 PickOrientation : pkGenlProc);
```

Purpose

Initialize help window and open help file.

Description

This constructor instantiates a help window using the default window options (DefWindowOptions), the default color set (DefaultColorSet), the default pick options (DefPickOptions), and the default help options (DefHelpOptions). See InitDeluxe for more details.

Example

```
if not PH.Init(9, 8, 72, 18,              {Window coordinates}
               'ENTRY.HLP',               {Help file}
               PickVertical)              {Pick Orientation}
then begin
  WriteLn('Failed to Init PagedHelpWindow,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PH (a PagedHelpWindow) using default window options and colors. It will read an external help file named ENTRY.HLP and use vertical pick orientation.

See Also

InitDeluxe InitMem

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte;
                       var Colors : ColorSet;
                       Options : LongInt;
                       HelpFile : PathStr;
                       PickOrientation : pkGenlProc);
```

Purpose

Initialize help window and open help file.

Description

This constructor instantiates a help window using the specified window Options, the specified Colors, default pick options (DefPickOptions), and default help options (DefHelpOptions). See InitDeluxe for more details.

Example

```
if not PH.InitCustom(9, 8, 72, 18,                    {Window coordinates}
                     DefaultColorSet,                 {ColorSet}
                     DefWindowOptions or wBordered,   {Window options}
                     'ENTRY.HLP',                     {Help file}
                     PickVertical)                    {Pick Orientation}
then begin
  WriteLn('Failed to Init PagedHelpWindow,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PH using a bordered window and default color options. It will read the external file ENTRY.HLP and use vertical pick orientation.

See Also

InitDeluxe InitMemCustom

InitDeluxe

Declaration

```
constructor InitDeluxe(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; HelpFile : PathStr;
                       PickOrientation : pkGenlProc; HelpOptions : Word);
```

Purpose

Initialize help window and open help file.

Description

This is the most general constructor for a help window that reads the help data from an external file. All of the related help window constructors make specific calls to this one.

X1, Y1, X2, and Y2 specify the active window coordinates within which the help window text and topic index will appear.

InitDeluxe plucks certain fields out of the specified ColorSet to use while drawing the help window. Besides those used by windows in general and by the PickList, InitDeluxe stores the following fields within the help window instance:

```
TextColor, TextMono             normal help text
FlexAHelpColor, FlexAHelpMono   help text marked with ^A
FlexBHelpColor, FlexBHelpMono   help text marked with ^B
FlexCHelpColor, FlexCHelpMono   help text marked with ^C
UnselXrefColor, UnselXrefMono   unselected cross-reference text
SelXrefColor, SelXrefMono       selected cross-ref text (PagedHelpWindow only)
BlockColor, BlockMono           marked block (ScrollingHelpWindow only)
```

Options are the bit-mapped flags that apply to all RawWindow-derived objects, such as wBordered. See the RawWindow Constants section for more details. InitDeluxe forces the wUserContents flag on, since the help window provides an UpdateContents method that can regenerate the window's contents on request.

HelpFile is the DOS name of the file containing the help text. HelpFile must specify the file's extension name; no default is provided. If HelpFile does not specify a drive or directory name, InitDeluxe will first look in the current directory, then in the directory of the program's EXE file (DOS 3.0 or later only), and finally in the directories specified by the DOS PATH. HelpFile must have been created by the MAKEHELP compiler; it must start with the HelpId signature string. InitDeluxe opens the file in the mode specified by the constant HelpFileOpenMode.

PickOrientation specifies the pick list orientation of the master topic index. It can take one of the values PickVertical, PickHorizontal, or PickSnaking. See "Pick List Orientations" in the introduction to the PickList object for details.

HelpOptions provides options specific to a help window. See the Constants declarations for details.

Note that InitDeluxe applies the default pick options (DefPickOptions) to the help window's pick mode. However, it does force the pkMinHeight option off in order to prevent the PickList constructor from modifying the specified window coordinates.

The HelpCommands command processor is assigned to the help window. The window is assigned a unit code of ucHelp (for the purposes of identifying it to error and help handlers). For a PagedHelpWindow, the hardware cursor is set to a default mode of cuHidden; for a ScrollingHelpWindow, the cursor is set to cuNormal.

InitDeluxe allocates several blocks of heap space:

- help header, SizeOf(HelpHeader) bytes
- an untyped file variable, SizeOf(File) bytes
- decompression buffer, BiggestTopic bytes
- if the hwStaticNameBuffer option is enabled, buffers for loading the topic names

See the discussion of the "Organization of a Help File" above for more information.

If the constructor fails, check the InitStatus variable for a detailed error code.

Example

```
if not PH.InitDeluxe(9, 8, 72, 18,                {Window coordinates}
                     DefaultColorSet,             {ColorSet}
                     DefWindowOptions or wBordered, {Window options}
                     'ENTRY.HLP',                 {Help file}
                     PickVertical,                {Pick Orientation}
                     hwStaticNameBuffer)          {Help options}
then begin
  WriteLn('Failed to Init PagedHelpWindow,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates PH using a bordered window and default color options. It will read the external file ENTRY.HLP and use vertical pick orientation. The topic name buffers will be allocated and loaded once. The hwPickIndex option, which is on by default, will be disabled so that users cannot access the master topic index while within help mode.

See Also

PickList.InitDeluxe

InitHelpFrame / InitMem / InitMemCustom

Declaration

```
procedure InitHelpFrame;
```

Purpose

Make help frame look like wFrame.

Description

A help window contains three Frame objects: wFrame, aFrame, and hwFrame. wFrame is displayed when the window is in pick mode, hwFrame when it is in help mode, and aFrame when the wAltFrame flag is set and the window is active but not current. You can independently initialize all of these. In many cases, wFrame and hwFrame will be identical with just a few differences (hwFrame might have a "more header" while wFrame might not, for example). When you call InitHelpFrame, OPHELP clones the current wFrame object onto hwFrame. You can then continue by adding more features to either object.

You should not use Frame.fCopy to perform the same action. InitHelpFrame sets an internal flag that OPHELP uses elsewhere.

Example

```
PH.Init(5, 8, 75, 18, 'ENTRY.HLP', PickVertical);
{...}
wOptionsOn(wBordered);
InitHelpFrame;
{...}
```

The help window is instantiated with default window options, which means it has no border. A border is added later with the wOptionsOn method. InitHwFrame synchronizes the help mode frame with the pick mode frame.

Declaration

```
constructor InitMem(X1, Y1, X2, Y2 : Byte; HelpAddr : Pointer;
                    PickOrientation : pkGenlProc);
```

Purpose

Initialize help window using help bound into code.

Description

This constructor instantiates a help window using the default window options (DefWindowOptions), the default color set (DefaultColorSet), the default pick options (DefPickOptions), and the default help options (DefHelpOptions). See InitMemDeluxe for more details.

See Also

InitMemDeluxe

Declaration

```
constructor InitMemCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                          Options : LongInt; HelpAddr : Pointer;
                          PickOrientation : pkGenlProc);
```

Purpose

Initialize help window using help bound into code.

Description

This constructor instantiates a help window using the specified window Options, the specified Colors, default pick options (DefPickOptions), and default help options (DefHelpOptions). See InitMemDeluxe for more details.

See Also

InitMemDeluxe

Declaration

```
constructor InitMemDeluxe(X1, Y1, X2, Y2 : Byte;
                          var Colors : ColorSet;
                          Options : LongInt;
                          HelpAddr : Pointer;
                          PickOrientation : pkGenlProc;
                          HelpOptions : Word);
```

Purpose

Initialize help window using help bound into code.

Description

This constructor works like InitDeluxe, except that the help file is bound into the program's EXE file rather than in an external file. As a result, there is no file access at all and the help logic is faster, simpler, and less prone to system errors.

The only heap space allocated by InitMemDeluxe is the decompression buffer, large enough for BiggestTopic. The other heap buffers allocated by InitDeluxe are managed instead by pointing directly to the bound-in file.

Example

```
var
  PH : PagedHelpWindow;
  Finished : Boolean;

  {$L ENTRYHLP.OBJ}
  procedure EntryHlp; external;

begin
  if not PH.InitMemDeluxe(9, 8, 72, 18,              {Window coordinates}
                          DefaultColorSet,           {ColorSet}
                          DefWindowOptions or wBordered,  {Window options}
                          @EntryHlp,                 {Address of help data}
                          PickVertical,              {Pick Orientation}
                          hwPickIndex+hwClearStackAtExit) {Help options}

  then begin
    WriteLn('Failed to Init PagedHelpWindow,  Status = ', InitStatus);
    Halt;
  end;
  {...}
  PH.Done;
end.
```

This program binds the help file in at compile time. The ENTRYHLP.OBJ file was created with Borland's BINOBJ utility. The following DOS command line was used:

```
BINOBJ ENTRY.HLP ENTRYHLP.OBJ ENTRYHLP
```

See the introductory section "Binding Help into a Program" for more information.

See Also

InitDeluxe

ItemString

Declaration

```
procedure ItemString(Item : Word; Mode : pkMode; var IType : pkItemType;
                     var IString : String); virtual;
```

Purpose

Supplies each item string when the master topic index is displayed or searched.

Description

This virtual method overrides the PickList's default version in order to provide formatted help topic names for the master topic index.

You can override the method again to customize the master topic index. The ENTRY demonstration program, for example, overrides it to highlight certain items in the index. See the definition of CustomHelpWindow in ENTRY.PAS.

See Also

PickList.ItemString

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a help system from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to the Store method of the help window. Load first calls PickList.Load to read the underlying PickList. Then it reads data specific to the help window.

Note that a help window stream does not store the help file itself. It stores only the description of the window, its headers, and configuration. It also stores either the name of the help file (name only, no path) or the pointer code of the bound-in help.

The stream registration procedure for PagedHelpWindow is PagedHelpWindowStream, and for ScrollingHelpWindow is ScrollingHelpWindowStream. There are also one or more procedure pointers to be registered independently. The example program below demonstrates how. See Store for more details.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpHelp;
var
  PH : PagedHelpWindow;
  S : BufIdStream;
begin
  {Load PH from a stream}
  S.Init('HELP.STM', SOpen, 1024);
  S.RegisterHier(PagedHelpWindowStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.Get(PH);
  S.Done;

  {Try out the help window}
  PH.SetTopic(0);
  PH.Process;
  if PH.GetLastCommand <> ccQuit then begin
    PH.SetTopic(PH.GetTopicChoice);
    PH.Process;
  end;
  PH.Done;
end.
```

Instantiates PH from the stream created in the Store example by calling the stream's Get method, which indirectly calls Load. Note that the address of the PickList's orientation procedure must be registered.

See BufIdStream (in OPROOT) for proper error checking using streams.

The only difference required to load a ScrollingHelpWindow is to refer to ScrollingHelpWindowStream rather than PagedHelpWindowStream.

See Also

PickList.Load Store

PickTopics / Process

Declaration
```
function PickTopics : Word;
```

Purpose
Return number of help index topics in help system.

Description
This returns the number of items that will appear in the master topic index.

Example
```
if PH.PickTopics < 25 then
  PH.hwOptionsOn(hwStaticNameBuffer);
```
Checks the number of indexed topics in this help file; and, if it is a relatively small number, requests the name buffers to be loaded permanently.

See Also
HighestTopic

Declaration
```
procedure Process; virtual;
```

Purpose
Draw help index or current topic.

Description
The help window Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Paged Help Window Commands" and "Scrolling Help Window Commands" for the PagedHelpWindow and ScrollingHelpWindow, respectively.

Process takes the topic to display from the top of the topic stack. (This topic may have been pushed there by a call to SetTopic or left there by a previous call to Process.) If the stack is not empty and the topic on top of the stack is not PickTopic (0), Process activates help mode, loads and decompresses the topic, and handles commands specific to help mode. Otherwise, Process activates the window's pick mode, loads the name buffers (if not already loaded), and calls PickList.Process.

Process may call several user-defined routines. The nested cross-reference function is specific to the help window. The premove procedure and the string search function are specific to the PickList. The error handling procedure is inherited from the CommandWindow. And the context-sensitive and keyboard background processing procedures are inherited from the CommandProcessor. (Note that it *is* possible to provide context-sensitive help, even for a help window. Process cannot be re-entered, however.)

Examples
```
PH.SetTopic(100);
PH.Process;
```
Displays topic 100 and lets the user browse the help system.
```
PH.SetTopic(PickTopic);
PH.Process;
if PH.GetLastCommand = ccSelect then
  ChosenTopic := PH.GetTopicChoice;
```
Displays the master topic index, gets a choice, and stores it in the ChosenTopic variable.

For a complete program example, see the introduction to this section.

See Also
PickList.Process SetTopic

Declaration

```
function ScreenWordXY(XPos, YPos : Byte;
                      ScanBack, ScanForward : Boolean) : String;
```

Purpose

Parse word from screen at absolute (XPos, YPos).

Description

ScreenWordXY is useful for writing help systems that are context-sensitive by virtue of reading information from the underlying screen. The POPHELP utility supplied with **Object Professional** is an example of such a program. ScreenWordXY is defined in the AbstractHelpWindow object for convenience only; it has no direct connection to the object.

XPos and YPos specify the absolute screen coordinates where parsing should start. These coordinates are usually obtained from the current hardware cursor position or perhaps the mouse cursor position.

ScreenWordXY defines a "word" as any contiguous sequence of characters found in the WordChars string, which defaults to contain the characters 'A'..'Z', '_', and '0'..'9'. Because WordChars is a global typed constant interfaced by OPHELP, you can modify it to include different characters if necessary (up to 127 characters maximum).

If the character at (XPos, YPos) is not in WordChars and the ScanBack parameter is True, ScreenWordXY will scan towards column 1 of the specified row until it finds a qualifying character. If there is still no qualifying character and ScanForward is True, it will scan towards larger column numbers until it reaches ScreenWidth or finds a word character. Then it finds the longest qualifying string and returns that as the function result. If there is no word to parse, it returns the empty string.

Example

```
WhereXYAbs(X, Y);
T :=  PH.FindTopic(PH.ScreenWordXY(X, Y, True, False), DefaultTopicMatch);
if T = 0 then begin
  PH.SetTopic(PickTopic);
  PH.Process;
  PH.SetTopic(PH.GetTopicChoice);
end else
  PH.SetTopic(T);
PH.Process;
PH.Done;
```

Reads a single word at the current cursor location, scanning backward to find a word if needed. FindTopic compares that string against all help topics. If no topics match, T is set to zero and the index is displayed. If FindTopic does find a match, we specify that topic and display the help.

See Also

FindTopic

SetHelpPadSize / SetNestedFunc

Declaration

```
procedure SetHelpPadSize(Pad : Byte);
```

Purpose

Set number of blank characters at edges of help window.

Description

By default, help text leaves one background character showing at both edges of the help window. Call this method to change this value. Note that MAKEHELP assumes the default value of one by optionally wrapping the help text at two columns less than the !WIDTH directive specifies.

Example

```
PH.SetHelpPadSize(2);
```

Adds a blank margin two characters wide to both sides of the help window.

See Also

SetPickPadSize

Declaration

```
procedure SetNestedFunc(NF : hwNestedFunc);
```

Purpose

Set function to call for nested cross-reference commands.

Description

The function NF must be compiled under the FAR model (using the {$F+} compiler directive), must be global, and must match the following prototype declaration:

```
{$F+}
function MyNestedFunc(HPtr : AbstractHelpWindowPtr) : Boolean;
```

See the discussion of "The Nested Function Hook" earlier in this section for more information.

Example

```
{$F+}
function MyNestedFunc(HPtr : AbstractHelpWindowPtr) : Boolean;
begin
  FastText('Current topic:         '+HexW(HPtr^.CurrentTopic), 24, 1);
  FastText('Cross-reference topic: '+HexW(HPtr^.CurrentXrefTopic), 25, 1);
  MyNestedFunc := False;
end;
{$F-}
...
HelpCommands.AddCommand(ccNested, 1, Ord(^M), 0);
HelpExitCommands := [ccPrevTopic, ccNested];
PH.SetNestedFunc(MyNestedFunc);
...
```

Assigns the <Enter> keystroke to the ccNested command, which disables the normal ccSelect command. So that we can make a selection from the master topic index, we add the ccNested command to the dynamic set of exit commands. SetNestedFunc activates the MyNestedFunc function, which displays the current topic number and the topic associated with the current cross-reference link, whenever the user presses <Enter> in the help window. If no cross-reference is active, SetNestedFunc displays FFFF instead. MyNestedFunc returns False to indicate that the Process method should not exit.

See Also

CurrentXrefTopic

Declaration
```
procedure SetPickPadSize(PadSize : Byte);
```

Purpose
Set number of characters for padding of pick items.

Description
PadSize background characters will be concatenated to both ends of each name string when the master topic index is displayed.

Example
```
PH.SetPickPadSize(5);
```
Pads all topics in the master index with five blanks.

See Also
SetHelpPadSize

Declaration
```
procedure SetPrevTopicHotSpot(SpotNum : Byte; Posn : FrameCharType);
```

Purpose
Activate a hot spot for previous topic command.

Description
SpotNum is the code associated with a hot spot previously added to hwFrame. For a single character spot added with the AddHotSpot method, the proper value for SpotNum is hsSpot. For a more general hot range, the value would be hsRegion0, hsRegion1, etc., as specified by a call to AddHotRegion.

Posn is the frame-relative position used when the hot spot was added—a frame corner or frame edge.

SetPrevTopicHotSpot stores these parameters in data fields internal to the help window object. When the Process method detects a mouse click over a hot spot, it compares the description of that hot spot to the one specified here. For a match, it converts the mouse click into a ccPrevTopic command, which causes the help window to redisplay the last topic.

A given help window can have at most one previous topic hot spot.

Example
```
PH.hwFrame.AddHotSpot(frTL, #4, DefaultColorSet);
PH.SetPrevTopicHotSpot(hsSpot, frTL);
```
Adds a diamond-shaped hot spot to the top left corner of the help window frame, then activates that hot spot for selecting previous topics.

See Also
Frame.AddHotRegion Frame.AddHotSpot

SetTopic

Declaration

```
procedure SetTopic(Topic : Word);
```

Purpose

Specify which topic is displayed next, PickTopic for index.

Description

SetTopic pushes the specified topic number onto the previous topic stack. The next call to Process pops that topic number from the stack. If the topic was PickTopic (0), the master index is displayed and a choice obtained. If the topic was non-zero, Process loads and displays the help for that topic.

Note that multiple calls to SetTopic will push multiple topics onto the stack. The last pushed will be displayed first. The user can access the others with the ccPrevTopic command (mapped to <AltF1> by default).

SetTopic validates the specified topic number only by assuring that it doesn't exceed the HighestTopic of the current help file. If it does, SetTopic generates the error epNonFatal+ecNoHelpForTopic. If the number is in the valid range but there is no help for that topic, no error will be generated until Process attempts to load it.

Example

```
PH.SetTopic(PickTopic);
```

Sets the index as the next topic. The next time Process is called, the index will be displayed.

```
PH.SetTopic(PH.GetTopicChoice);
```

Sets the last selected topic (from a pick list) as the topic to be displayed. The next time Process is called, that topic will be displayed.

See Also

Process

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a help system in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the help window to the stream S. It starts by calling PickList.Store, then continues to write the data specific to a help window.

Note that a help window stream does not store the help file itself. It stores only the description of the window, its headers, and configuration. It also stores either the name of the help file (name only, no path) or the pointer code of the bound-in help.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

The stream registration procedure for PagedHelpWindow is PagedHelpWindowStream, and for ScrollingHelpWindow is ScrollingHelpWindowStream. There are also one or more procedure pointers to be registered independently. The example program below demonstrates how.

The orientation routine for the underlying PickList (PickVertical, PickHorizontal, or PickSnaking) must be registered. Failing to do so will result in the error epFatal+ecBadPickOrient when the stream is loaded.

The help window's stream registration procedure automatically registers the PickList's command handler (SingleChoice). Help windows use the virtual method approach to the user-defined string procedure, so there is no need to register that.

If the help file has been linked into the program's EXE file, the address of the help must be registered.

If any of the following user-written routines are being used, they must be registered or they won't be activated when the stream is reloaded. These include the nested function, block copy procedure (ScrollingHelpWindow only), premove procedure, and topic index search function.

If the help window uses any but the default CommandProcessor (HelpCommands), the address of that object must be registered. See CommandWindow.Store for an example.

Example

```
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpHelp;
var
  PH : PagedHelpWindow;
  S : BufIdStream;
begin
  {Initialize a help window}
  PH.InitCustom(9, 8, 72, 18, DefaultColorSet, DefWindowOptions+wBordered,
                'ENTRY.HLP', PickVertical);
  PH.AddTopicHeader(1, 60, heTC);
  PH.wFrame.AddHeader(' Topic Index ', heTC);
  PH.AddMoreHeader(' || for more ', heBR, #24, #25, '', 2, 3, 0);
  {Store PH to a stream}
  S.Init('HELP.STM', SCreate, 1024);
  S.RegisterHier(PagedHelpWindowStream);
  S.RegisterPointer(ptPickVertical, @PickVertical);
  S.Put(PH):
  S.Done;
  PH.Done;
end.
```

Store

Instantiates PH, adds some customizations and stores PH to a stream. See Load for an example program to load this stream. The only difference required to store a ScrollingHelpWindow is to refer to ScrollingHelpWindowStream rather than PagedHelpWindowStream.

See BufIdStream (in OPROOT) for proper error checking using streams.

If the help file had been bound into the program, we'd need to add a statement like

```
S.RegisterPointer(2000, @EntryHlp);
```

to the registration sequence. 2000 is the pointer registration code, which can be any number 1000 or larger that is unique within the stream. EntryHlp is the name of the dummy procedure used to link in the help file (see InitMemDeluxe for a related example).

See Also

Load PickList.Store

Declaration

```
function TopicStackPtr : StaticQueuePtr;
```

Purpose

Return a pointer to the topic stack.

Description

Each help window contains an instance of a StaticQueue object used to manage the stack of previously displayed topics. TopicStackPtr returns its address so that you can apply the various methods of the StaticQueue type to it.

Examples

```
if PH.TopicStackPtr^.Elements = 0 then
  {nothing is on the stack} ;
```

Checks to see whether any topics are on the stack.

```
var
  StackRec : phStackRec;
...
PH.TopicStackPtr^.PeekTail(StackRec);
if StackRec.STopic = 0 then
  {top element on stack is a pick request, remove it}
  PH.TopicStackPtr^.PopTail(StackRec);
```

Peeks at the top record on the stack of a PagedHelpWindow. If the topic number is zero, it is popped from the stack.

See Also

StaticQueue.Elements StaticQueue.PeekTail

Declaration

```
procedure UpdateContents; virtual;
```

Purpose

Redraw the help window in pick or help mode.

Description

You shouldn't need to call UpdateContents directly. Call Process instead.

UpdateContents performs the following operations:

If the window is in help mode and a topic has been loaded previously, UpdateContents draws it and updates the headers and scroll bars. If the window is in pick mode, UpdateContents loads the name buffers if necessary and calls PickList.UpdateContents. If the name buffers weren't allocated previously, UpdateContents deallocates them before exiting.

See Also

Draw Process
PickList.UpdateContents

2. PagedHelpWindow: Help by the Page

The behavior of PagedHelpWindow is much like the TPHELP unit in Turbo Professional 5.0. Scrolling occurs by page only; there is no horizontal scrolling support. PagedHelpWindow inherits all of the functionality just described for AbstractHelpWindow. What it implements is the logic to display the help text and to manage the movement of a highlight bar from one cross-reference to another. Because PagedHelpWindow provides no new public methods, there are no method descriptions in this section.

Paged Help Window Commands

The PagedHelpWindow Process method offers appropriate commands to browse through the help text and select cross-references, to back up through the topic stack, and to make selections from the master topic index. The list below arranges these commands by category. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant HelpKeySet (in OPHELP.ICD).

Help Page Scrolling

ccPageUp **<PgUp>, <CtrlR>**
Display previous help page of current topic.

ccPageDn **<PgDn>, <CtrlC>**
Display next help page of current topic.

ccHome, ccTopOfFile **<Home>, <CtrlQ><S>, <CtrlPgUp>, <CtrlQ><R>**
Display first help page of current topic.

ccEnd, ccEndofFile **<End>, <CtrlQ><D>, <CtrlPgDn>, <CtrlQ><C>**
Display last help page of current topic.

Cross-Reference Highlight Bar Movement

ccUp, ccScrollUp **<Up>, <CtrlE>, <CtrlW>**
Move highlight bar to previous cross-reference in the vertical direction.

ccDown, ccScrollDn **<Down>, <CtrlX>, <CtrlZ>**
Move highlight bar to next cross-reference in the vertical direction.

ccLeft, ccPrevXref **<Left>, <CtrlS>, <ShiftTab>**
Move highlight bar to previous cross-reference horizontally. Wraps to end of previous row when the highlight bar reaches the leftmost cross-reference.

ccRight, ccNextXref **<Right>, <CtrlD>, <Tab>**
Move highlight bar to next cross-reference horizontally. Wraps to start of next row when the highlight bar reaches the rightmost cross-reference.

Other

ccSelect **<Enter>**
If there is a highlight bar visible, load the help topic indicated by its cross-reference and display the first page of that topic. If the topic number indicated by the cross-reference is zero, switch to pick mode and display the Master topic index. If the window is in pick mode already, load the help topic indicated by the current pick item and display its first page.

ccMouseSel **<ClickLeft>**
Move the highlight bar to the cross-reference indicated by the mouse, if any. If the highlight bar is already over the indicated cross-reference, act as if the ccSelect command were entered there. This command can also be used to scroll the help window by clicking on a scroll bar, or to reload the previous topic by clicking on a previous topic hot spot (see SetPrevTopicHotSpot). When the help window is in pick mode, the mouse acts as described for pick list commands.

ccPrevTopic **<AltF1>**
Load the help topic, if any, loaded prior to the current one, display the page most recently shown, and highlight the most recent cross-reference. This command may also be activated by a previous topic hot spot.

ccIndex **<F1>**
If the hwPickIndex option is enabled, switch the window into pick mode and display the master topic index.

ccNested **no keys assigned**
Call the user-written nested function. See the discussion of this function earlier in this section.

ccQuit **<Esc>, <ClickRight>**
Exit the help window or master topic index.

ccHelp **<ClickBoth>**
Help. If a user-written help routine has been established by calling SetHelpProc, this command will call that routine; otherwise this command does nothing. The help routine is passed the following three parameters: ucHelp, @Self, and a help topic number designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0. Note that the AbstractHelpWindow.Process method will not allow reentry; the ccHelp command cannot generate a call that leads into the same OPHELP object.

Types

```
Pages = 0..MaxPagesPerSection;
```

Defines the valid range of page numbers for each topic. Zero is reserved for internal use.

```
phStackRec =
  record
    STopic : Word;          {Which topic}
    SPage  : Pages;         {Page in topic}
    SXnum  : Xrefs;         {Xref item selected}
  end;
```

Defines the information stored on the topic stack for each previous topic.

```
phPageRec =                 {Display state at the start of a given page}
  record
    TextOfs : Word;         {Offset into decompressed text}
    FlexSP  : Byte;         {Flex stack pointer}
    FlexStk : FlexStack;    {Stack of active attribute indexes}
  end;
phPageArray = array[1..MaxPagesPerSection] of phPageRec;
```

Information used internally by the PagedHelpWindow to scroll quickly from page to page of a topic. The phPageArray is initialized when the current topic is loaded.

```
phXrefRec =                    {Describes a given cross-reference}
  record
    XrefPage  : Pages;         {Which page the xref displays on}
    XrefRow   : Byte;          {Which row of window}
    XrefCol   : Byte;          {Which col of window}
    XrefOfs   : Word;          {Offset in uncompressed text buffer}
    XrefTopic : Word;          {Which topic is cross-referenced}
  end;
phXrefArray = array[1..MaxXrefsPerSection] of phXrefRec;
```

Information used internally by the PagedHelpWindow to move the highlight bar quickly from cross-reference to cross-reference. The phXrefArray is initialized when the current topic is loaded.

```
PagedHelpWindowPtr = ^PagedHelpWindow;
PagedHelpWindow =
  object(AbstractHelpWindow)
    ...
  end;
```

The paged help window object.

3. ScrollingHelpWindow: Help by the Line

Besides offering the ability to scroll horizontally and line-by-line, ScrollingHelpWindow adds commands for block marking and copying. This new capability was important in developing the POPHELP online help engine. By combining OPHELP's block copy hook with the OPREPLAY unit, POPHELP transfers selected portions of the help text to the underlying application, usually a text editor. Performing this feat in a swappable TSR like POPHELP is a difficult exercise; we refer you to the POPHELP source code to see how it's done. When a help system is integrated within a normal application, the paste procedure can be substantially simpler, as we'll show below.

In the ScrollingHelpWindow, a cross-reference is selected by moving the hardware cursor over its marking text and then pressing <Enter>. Cross-references are always drawn in the unselected attribute (see AbstractHelpWindow.InitDeluxe for more information).

Scrolling Help Window Commands

The ScrollingHelpWindow Process method offers appropriate commands to browse through the help text and select cross-references, to back up through the topic stack, and to make selections from the master topic index. The list below arranges these commands by category. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant HelpKeySet (in OPHELP.ICD). Note that PagedHelpWindow and ScrollingHelpWindow share the same CommandProcessor and key set.

Cursor Movement and Scrolling

In pick mode, all relevant commands work as described for the PickList.

ccUp **<Up>, <CtrlE>**
Cursor up one line, scrolling the help window when the cursor reaches the top of the window.

ccScrollUp **<CtrlW>**
Scroll the help text down one line.

ccDown **<Down>, <CtrlX>**
Cursor down one line, scrolling the help window when the cursor reaches the bottom of the window.

ccScrollDn **<CtrlZ>**
Scroll the help text up one line.

ccLeft **<Left>, <CtrlS>**
Cursor left one column. Scroll the help window one column if the cursor reaches the left edge of the window and column 1 of the text is not currently visible.

ccRight **<Right>, <CtrlD>**
Cursor right one column. Scroll the help window one column if the cursor reaches the right edge of the window and the text hasn't scrolled to the maximum horizontal scroll limit yet.

ccPageUp **<PgUp>, <CtrlR>**
Scroll the help text down by the window's height.

ccPageDn **<PgDn>, <CtrlC>**
Scroll the help text up by the window's height.

ccHome **<Home>, <CtrlQ><S>**
Cursor to column 1 of the current row.

ccTopOfFile **<CtrlPgUp>, <CtrlQ><R>**
Cursor to row 1, column 1 of the help text, scrolling it if necessary.

ccEnd **<End>, <CtrlQ><D>**
Cursor to the end of the current line of help text.

ccEndofFile **<CtrlPgDn>, <CtrlQ><C>**
Cursor to the end of the last line of help text in the current topic, scrolling the text if necessary.

ccScreenTop **<CtrlHome>, <CtrlQ><E>**
Cursor to the first row of the help window without scrolling the text or changing the cursor column.

ccScreenBot **<CtrlEnd>, <CtrlQ><X>**
Cursor to the last row of the help window without scrolling the text or changing the cursor column.

ccWordLeft **<CtrlLeft>, <CtrlA>**
Cursor left one word. If the cursor is at the beginning of a line, it is moved to the end of the previous line.

ccWordRight **<CtrlRight>, <CtrlF>**
Cursor right one word. If the cursor is at the end of a line, it is moved to the beginning of the following line.

ccPrevXref **<ShiftTab>**
Cursor to the previous cross-reference, scrolling the window if necessary. If there are no previous cross-references, do nothing.

ccNextXref **<Tab>**
Cursor to the next cross-reference, scrolling the window if necessary. If there are no more cross-references, do nothing.

Block Operations

The block operations are active only if the hwBlockMark option is enabled.

ccBlkBegin **<F7>, <CtrlK><B>**
Mark the current cursor location as the start of a block. If the begin marker is before the end marker, the block will be highlighted.

ccBlkEnd **<F8>, <CtrlK><K>**
Mark the current cursor location as the end of a block. Note that the last highlighted character is the one just before the cursor. If the cursor is beyond the end of a line, the block will end at the last non-blank character of the line.

ccBlkToggle **<CtrlK><H>**
Toggle the highlighting of a marked block.

ccBlkCopy **<CtrlK><C>**
If a valid block is marked and visible, call the user-written block copy function. Otherwise, ignore the command.

Other

ccSelect **<Enter>**
If the cursor is over a cross-reference marker, load and display the indicated help topic. If the topic number indicated by the cross-reference is zero, switch to pick mode and display the master topic index. If the window is in pick mode already, load and display the help topic indicated by the current pick item.

ccMouseSel **<ClickLeft>**
Move the cursor to the position indicated by the mouse. If the cursor is already over a cross-reference and ccMouseSel is executed a second time, act as if the ccSelect command were entered there. This command can also be used to scroll the help window by clicking on a scroll bar, or to reload the previous topic by clicking on a previous topic hot spot (see SetPrevTopicHotSpot). When the help window is in pick mode, the mouse acts as described for pick list commands.

ccPrevTopic **<AltF1>**
Load the help topic, if any, loaded prior to the current one and scroll the text to the position most recently shown. This command may also be activated by a previous topic hot spot.

ccIndex **<F1>**
If the hwPickIndex option is enabled, switch the window into pick mode and display the master topic index.

ccNested **no keys assigned**
Call the user-written nested function. See the discussion of this function earlier in this section.

ccQuit **<Esc>, <ClickRight>**
Exit the help window or master topic index.

ccHelp **<ClickBoth>**
Help. If a user-written help routine has been established by calling SetHelpProc, this command will call that routine, passing it the following three parameters: ucHelp, @Self, and a help topic number designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0. Note that the AbstractHelpWindow.Process method will not allow reentry; the ccHelp command cannot generate a call that leads into the same OPHELP object.

Using the Block Copy Hook

If the hwBlockMark option has been enabled, a block has been marked, and the user executes the ccBlkCopy command, OPHELP calls a user-written function known as the block copy hook. This function must be compiled with the FAR model (using the {$F+} directive), must be global, and must have parameters that match the following prototype definition:

```
{$F+}
function MyCopyFunc(T : CharArrayPtr; Len : Word;
                    SH : ScrollingHelpWindowPtr) : Boolean;
```

When OPHELP calls this function, it passes the address of the first marked character in the parameter T, the number of marked characters in Len, and the address of the help window object in SH. MyCopyFunc is responsible for transferring the characters elsewhere: to a file, to another window, to the keyboard buffer. If desired, it can reset or hide the block, update the screen, and cause the Process method to exit. If MyCopyFunc returns True, Process will exit with the ccBlkCopy exit command; if it returns False, Process will loop around to get further commands.

To assign control to your copy function, you use the SetSendFunc method:

```
procedure SetSendFunc(SF : shSendFunc);
  {-Set function to be called by CopyBlock command}
```

Copying the characters from the buffer at T^ may be a little more complex than you expect. Although the buffer has been decompressed and is stored in "plain text," it still contains various control characters used for varying the video attributes and implementing cross-reference links. It's the copy function's responsibility to filter these characters or translate them.

Here's an example program that appends marked help information to a text file.

```
program HelpBlockExample;  {EXHELP2.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpMouse, OpFrame, OpWindow, OpPick, OpHelp;
const
  HelpFile = 'ENTRY.HLP';
  BlockFile = 'ENTRY.BLK';
var
  SH : ScrollingHelpWindow;

  {$F+}
  function MyCopyFunc(T : CharArrayPtr; Len : Word;
                      SH : ScrollingHelpWindowPtr) : Boolean;
  var
    I : Word;
    F : Text;
  begin
    {Don't exit the Process method}
    MyCopyFunc := False;

    {$I-}
    {Append to the block file}
    Assign(F, BlockFile);
    Append(F);
    if IoResult <> 0 then
      Rewrite(F);
    {$I+}

    {Scan the help information}
    I := 0;
    repeat
      case T^[I] of
        Attr1Toggle..Attr3Toggle,
        XrefToggle :       {Modifying video attribute}
          ;

        IndexMarker :      {Cross-reference topic follows}
          Inc(I, 2);

        LineBrkMark,       {End of line}
        PageBrkMark :      {End of page--acts like end of line here}
          WriteLn(F);

      else
        {Normal text}
        Write(F, T^[I]);
      end;
      Inc(I);
    until I >= Len;
```

```
    WriteLn(F);
    Close(F);

    {Reset the block and update the window}
    SH^.HideBlock;
    SH^.Draw;
  end;

begin
  if not SH.InitDeluxe(9, 8, 72, 18,
                       DefaultColorSet,
                       DefWindowOptions or wBordered,
                       HelpFile,
                       PickVertical,
                       hwStaticNameBuffer+hwPickIndex+hwBlockMark)
  then begin
    WriteLn('Error initializing help window: ', InitStatus);
    Halt;
  end;

  {Add some features}
  SH.AddTopicHeader(1, 60, heTC);
  SH.wFrame.AddHeader(' Topic Index ', heTC);
  SH.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  SH.hwFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  SH.hwFrame.AddScrollBar(frBB, 0, MaxLongInt, DefaultColorSet);

  SH.SetBlockAttr($70, $70);
  SH.SetSendFunc(MyCopyFunc);

  {Display topic 1}
  SH.SetTopic(1);
  SH.Process;

  {Deallocate help window}
  SH.Done;
end.
```

MyCopyFunc will never cause the Process method to exit in this example, so it sets its result to False. It opens the output file and attempts to append to it; if the file doesn't already exist, it creates it. (The I/O error checking in this example is barely adequate; a real program would do more.)

The repeat loop shows how to scan the help text. Index position 0 of the array pointed to by T always has the first character of the marked block. In this example, T^[I] refers to the next character to interpret. Attr1Toggle through Attr3Toggle and XrefToggle are control characters that we can ignore here. IndexMarker is another control character. We don't want to write it to the text file, but we must always skip the two characters that follow it since they are the (binary format) topic number of the cross-reference link. LineBrkMrk and PageBrkMark indicate the end of a line, so we WriteLn to the text file. We write any other character to the output file. The loop continues until the index I equals or exceeds the specified Len of the block.

Once all the characters have been written, we terminate the output block with a WriteLn statement and close the file. The copy procedure ends by calling two of the ScrollingHelpWindow's methods: HideBlock hides the just-copied block so it can't be copied immediately again, and Draw updates the help window to show this change. Note that the copy function could call any other ScrollingHelpWindow methods as long as the window remains current with its coordinates unchanged when the function returns.

The main body of the program activates the block commands by specifying the hwBlockMark option to the window's constructor. It sets the block marking attribute to reverse video ($70) for both color and monochrome monitors, and calls SetSendFunc to specify that MyCopyFunc will be called for the ccBlkCopy command.

Types

```
shStackRec =
  record
    STopic : Word;          {Which topic}
    First  : shLineRec;     {State of first line}
    Last   : shLineRec;     {State of last line}
    HorOfs : Word;          {Horizontal screen offset}
    CursX  : Byte;          {Cursor X position}
    CursY  : Byte;          {Cursor Y position}
  end;
```

Defines the information stored on the topic stack for each previous topic.

```
shLineRec =
  record
    TextOfs : Word;         {Offset into decompressed text}
    FlexSP  : Byte;         {Flex stack pointer}
    FlexStk : FlexStack;    {Stack of active attribute indexes}
  end;
```

Information used internally by the ScrollingHelpWindow to scroll quickly from line to line of a topic.

```
shSendFunc = function (T : CharArrayPtr ; Len : Word;
                       SH : ScrollingHelpWindowPtr) : Boolean;
```

Procedure type for the block copy hook. See the discussion above for more information.

```
ScrollingHelpWindowPtr = ^ScrollingHelpWindow;
ScrollingHelpWindow =
  object(AbstractHelpWindow)
    ...
  end;
```

The scrolling help window object.

Declaration
```
procedure HideBlock;
```

Purpose
Hide the current block. No screen update.

Description
This method will typically be called inside of a user-written copy function. It simply resets an internal flag that indicates that a block might be visible. If block commands have not been activated, HideBlock does nothing. Call Draw to update the screen.

Example
```
SH^.HideBlock;
```
Hides the block for a ScrollingHelpWindow whose address is SH. See "Using the Block Copy Hook" for a complete example.

Declaration
```
procedure ResetBlock;
```

Purpose
Clear the current block. No screen update.

Description
Works like HideBlock, with the addition that it resets the internal block begin and end positions to zero.

Declaration
```
procedure SetBlockAttr(Color, Mono : Byte);
```

Purpose
Set the attribute for blocks.

Description
The default block attributes are the BlockColor and BlockMono fields of the ColorSet used to initialize the help window. SetBlockAttr changes the attributes to something else.

Example
```
SH.SetBlockAttr($02, $70);
```
Sets the block marking attribute to green on black for a color system and reverse video for monochrome.

See Also
AbstractHelpWindow.InitDeluxe

SetMaxScroll / SetSendFunc

Declaration

```
procedure SetMaxScroll(MaxFirstCol : Word);
```

Purpose

Set a new value for maximum horizontal scrolling.

Description

If the hwAdjustScroll option is enabled, the maximum horizontal scrolling column is set to allow the longest line in the help file to just appear in the current help window width. If this option is disabled, horizontal scrolling is essentially unlimited. Use SetMaxScroll to limit the scrolling in this case. If MaxFirstCol is 1, no horizontal scrolling is allowed.

Example

```
SH.SetMaxScroll(50);
```

Help will scroll horizontally by up to 49 characters.

Declaration

```
procedure SetSendFunc(SF : shSendFunc);
```

Purpose

Set function to be called by ccBlkCopy command.

Description

The function SF must be compiled with the FAR model (using the {$F+} directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
function MyCopyFunc(T : CharArrayPtr; Len : Word;
                    SH : ScrollingHelpWindowPtr) : Boolean;
```

See "Using the Block Copy Hook" for more information and a complete example.

See Also

HideBlock

4. Procedures and Functions

Besides the stream registration procedures and default user hook routines already mentioned, OPHELP interfaces one function for detecting the "object-preference" of a help file. Some help files may have been designed to work better with a ScrollingHelpWindow; others with a PagedHelpWindow. Help file designers can indicate a preference by using the !SCROLLING directive. This detection capability is probably useful only to a utility like POPHELP, which may need to read help files written by any number of people. For a normal application, you'll know what kind of help file you have in advance.

ScrollingHelpFile

Declaration

```
function ScrollingHelpFile(HelpFile : PathStr) : Boolean;
```

Purpose

Return True if HelpFile is designed for a scrolling help window.

Description

ScrollingHelpFile opens the specified help file in the same way that a help window constructor does: it looks for the specified pathname first; if that fails and HelpFile doesn't specify a drive or directory, it searches the directory of the program and the DOS PATH to find it. ScrollingHelpFile then opens the file (using HelpFileOpenMode), reads its header, and closes the file again. If the Scrolling flag is enabled in the header, ScrollingHelpFile returns True. If the flag is not set, or if any error occurs while opening or reading the help file, ScrollingHelpFile returns False.

Example

```
program PageOrScrollExample;  {EXHELP3.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpMouse, OpFrame, OpWindow, OpPick, OpHelp;
const
  HelpFile = 'ENTRY.HLP';
var
  H : AbstractHelpWindowPtr;
  Scrolling : Boolean;
begin
  {Determine if help file prefers scrolling}
  Scrolling := ScrollingHelpFile(HelpFile);
  {Allocate a help window of the preferred type}
  if Scrolling then
    H := New(ScrollingHelpWindowPtr,
             InitDeluxe(9, 8, 72, 18, DefaultColorSet,
                        DefWindowOptions or wBordered,
                        HelpFile, PickVertical,
                        hwStaticNameBuffer+hwPickIndex))
  else
    H := New(PagedHelpWindowPtr,
             InitDeluxe(9, 8, 72, 18, DefaultColorSet,
                        DefWindowOptions or wBordered,
                        HelpFile, PickVertical,
                        hwStaticNameBuffer+hwPickIndex));
  if H = nil then begin
    WriteLn('Error initializing help window: ', InitStatus);
    Halt;
  end;
  {Add some features}
  H^.AddTopicHeader(1, 60, heTC);
  H^.wFrame.AddHeader(' Topic Index ', heTC);
  H^.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  H^.hwFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  if Scrolling then
    {Horizontal scroll bar for scrolling help window only}
    H^.hwFrame.AddScrollBar(frBB, 0, MaxLongInt, DefaultColorSet);
  {Display topic 1}
  H^.SetTopic(1);
  H^.Process;
  {Deallocate help window}
  Dispose(H, Done);
end.
```

In order to declare an object that can be initialized as either a paged or scrolling help window, we must define it as a pointer to a help window. ScrollingHelpFile tells us whether scrolling is desired. If so, we call the ScrollingHelpWindow's constructor; otherwise, the PagedHelpWindow's constructor. The call to New allocates the appropriate number of bytes for each object type. If the constructor succeeds, the wonders of polymorphism cause the same pointer to act as a scrolling or paged help window, even when we apparently use the same method calls. The only place where we check the scrolling flag again is when we decide to add a horizontal scroll bar; the PagedHelpWindow doesn't need or support a horizontal scroll bar.

See Also

AbstractHelpWindow.InitDeluxe

H. Utility and Sample Programs

This section covers three programs: MAKEMENU, a utility for designing menu systems interactively; PSCREEN, a memory-resident program that captures text screens and saves them to disk files; and WINWOW, a flashy example showing some of the special effects available from the OPWINDOW unit. Note that MAKEHELP, another utility program used to compile help text for the OPHELP unit, has already been discussed in that section.

1. MAKEMENU

MAKEMENU is an interactive generator of hierarchical menu systems. With it, you can visually develop such systems, quickly and easily achieving proper positioning, colors, and appearance that would take much longer to write from scratch. After you are satisfied, MAKEMENU will write a stream-based library or generate Pascal source code that can be easily integrated into your own programs.

Although you can use the OPMENU unit without taking advantage of MAKEMENU, and although MAKEMENU cannot access every advanced feature of the Menu object, you will definitely want to learn how to use this utility. With a little experimentation, you could probably use it without even reading this documentation, but you'll probably find the going easier if you follow along here. If you've used MAKEMENU from Turbo Professional, this new version will seem very familiar; even so, it has a number of new features that you'll want to check out.

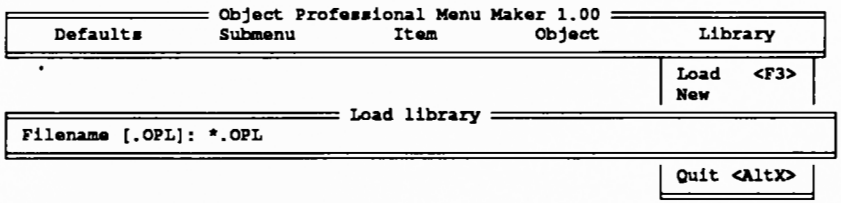
You'll find it easier to use this section if you've already read the introductory portions of the OPFRAME, OPWINDOW, and OPMENU documentation. We will assume that you are already familiar with windows, borders, submenus, menu items, and so on. Some familiarity with the concept of object libraries (implemented in OPROOT) will also be helpful.

Getting Started

To introduce you to MAKEMENU, we'll start by looking at a sample menu system that we've prepared for this purpose. For this exercise you'll need two files: MAKEMENU.EXE and SAMPLE.OPL. Assuming that they are both in the current directory, simply enter

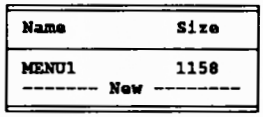
```
MAKEMENU
```

at the DOS command line. What you'll see is something like this:



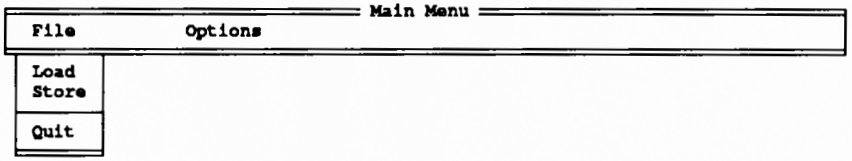
The Load item on the Library submenu has been pre-selected for you, and all you need to do now is enter the name of a new or existing library ("OPL" means **Object Professional** Library). We could enter "SAMPLE" at this prompt–a file extension of "OPL" is assumed–but instead let's just press <Enter>. This will bring up a directory list showing all the files with an extension of "OPL" in the current directory. Unless you have other OPL files in this directory, your only choice should be "sample.opl," which you can now select by pressing <Enter> again.

Now you'll be shown a list of all the menu systems in SAMPLE.OPL. There may be objects of other types in the library, and in this case there are, but you'll only see the names of the menu systems. The list should look something like this:



You have two choices here. You can create a new menu system by selecting "New", or you can load the existing object MENU1. Press <Enter> to do so.

When you select MENU1 from the list, the object will be displayed at the top of the screen, with the backdrop (block characters with polka-dots form the default backdrop) underneath it and the MAKEMENU menu system on top of it. Press <F10> to leave MAKEMENU's menu system and enter the Test mode. Press <Enter> once more and what you'll see is something like this:



This example menu system has a horizontally oriented top level menu with two items; each top level item has a drop-down submenu. The 'File' submenu has the three items shown. The 'Options' item has another two items, which can't be seen until their submenu is activated.

At the bottom of the screen is a line reminding you how to get back to the main menu or to exit MAKEMENU:

```
<F10> Return to main menu  <AltX> Exit program
```

You can use the arrow keys, <Esc> and <Enter> to browse around the example menu system. <F10> will take you back to the main menu. You'll also find that if you press <Esc> at the top level of the example menu, or <Enter> on any item that has no submenu, you'll be returned to the main menu.

This test mode gives you an opportunity to try out your menu system and determine how it will work when you plug it into your own program. It also gives you the opportunity to position the highlight bar within a particular submenu and on a specific item--MAKEMENU will use this position to establish the context of the commands you give to modify your menu system.

Now we're ready to try modifying the menu system a bit. To prepare, browse around the example menu until the highlight bar is over the File-Store item, then press <F10>. MAKEMENU's menu system will reappear.

Let's say that you want to change 'Store' to say 'Save' instead. To do so, select MAKEMENU's Item-Edit command. Because the example's highlight bar was over 'Store', MAKEMENU knows that that's the item to edit, and it will display a prompt box with 'Store' already displayed. Backspace over 'Store', type in 'Save', and press <Enter>. The example menu will be updated immediately to show the change.

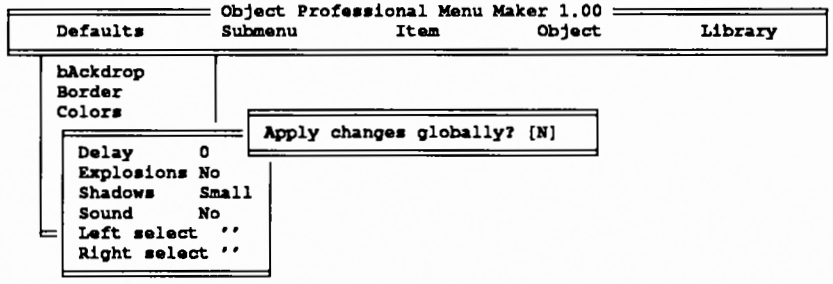
Note that if you have a Microsoft-compatible mouse, you can also use it within MAKEMENU. Pressing the left mouse button is equivalent to typing <Enter> and the right button is

equivalent to <Esc>. Clicking both buttons is the same as pressing <F10>. By moving the mouse around, you can select items from either MAKEMENU's menu or from the one you're designing (unless you have mouse support disabled there). The mouse can also drag menus or items around the screen--what it does will be intuitive and obvious when you use it.

Now that you've changed 'Store' to 'Save' you'll notice that the submenu has too much space on the right. Let's use MAKEMENU to correct that little problem by selecting Submenu-Resize. The main menu will disappear, leaving the cursor blinking in the lower right corner of the example. Press the <Left> arrow once to shrink the menu, then <Enter> to accept the result. It's as easy as that!

You've noticed by now that the example menu system has drop shadows on all the submenus. MAKEMENU lets you apply various special effects like this to each submenu individually or to all of them alike. Let's explore the various ways to change shadows. First, select MAKEMENU's Submenu-Settings-Effects-Shadows command. This menu choice lets you toggle between three settings: None (no shadows), Big, and Small. Try pressing <Enter> repeatedly to see what difference each setting makes. Stop when you get to 'Big' the second time. As you saw, the Submenu-Settings commands apply only to the currently selected submenu.

Now, select Defaults-Effects-Shadows and press <Enter>. MAKEMENU's menu will show the following:



If you enter <Y> in response to this prompt, MAKEMENU will modify every submenu in the entire example to have Small shadows. In this case, let's go ahead and press <Y>. You'll see that both the top level menu and the drop-down submenu are changed. The same thing happens to the other submenu, although you can't see it until you select that submenu. MAKEMENU's ability to perform global updates like this is very powerful, but you must use it cautiously since there is no undo command available--except for reading the menu system from the library again.

If instead of answering <Y> we had pressed <Enter> or <N>, you wouldn't have seen any change occur. In that case we would have changed just the default setting: the next time we added a new submenu it would receive a Small shadow. This distinction between global updates and default settings applies to most items under MAKEMENU's Defaults submenu.

Those are all the changes we're going to make to this menu system. We could save them right now by pressing <F2>, but that would make them permanent, and you might want to repeat this lesson. So instead let's save the modified menu system under a new name by selecting MAKEMENU's Object-Save as command. At the prompt for a name, enter 'MENU2'. You'll see a message saying that the object has been saved.

We're not quite done with this example, though, because we don't yet have anything we can use in a program. Select MAKEMENU's Object-Write source command. You'll then see a popup window asking you for a filename:

```
Name of source file [.PAS]: MENU2.PAS
```

Press <Enter> to select the default response. In a few seconds, you'll see a message indicating that your program has been generated. The result is a ready-to-run test program. To verify that it works, press <AltX> to exit from MAKEMENU, then use either compiler (TURBO or TPC, preferably the latter) to compile it to an EXE file. Run MENU2.EXE and you'll see the same menu system (over a black background, however), from which you can make one selection to return to the DOS prompt. You'll probably want to glance at the source code that was generated at this point, but that's up to you. We'll talk more about source code generation later.

Your First Menu System

As you've seen, modifying an existing menu system is quite easy. Creating a new one isn't really much harder, though, and that's what we'll be doing in this section.

The first step is to load a library. We've already learned how to load an existing library, so let's create a new one instead. When MAKEMENU starts, and you're asked for the name of a library, don't press <Enter> to get a directory listing. Enter a new name instead, say 'MYLIB'. MAKEMENU won't be able to find MYLIB.OPL, of course, so it will ask if you want to create it. Press <Y> or <Enter>. You'll then be asked for the number of directory entries in the library. This value determines how many objects can be stored in it. To accept the default response of 15, press <Enter>.

As before, you'll now see a list of menu systems to load. This time, however, the list will be empty, and your only choices will be to press <Esc> to cancel the operation or press <Enter> to indicate that you want to create a new one. When you press <Enter>, you'll be asked for a name for your menu system. Be careful here. Although you are allowed to enter a string of up to 12 characters, you will probably never want to enter one that long. The name specified here will be used as the name of your program when source code is generated, so it is wise to enter a string of no more than 8 characters that both DOS and the compiler will be happy with.

After you've entered a name, you'll see an empty window near the top left of the screen, waiting for you to add items to it. You don't need to worry about sizing this window yet, since MAKEMENU will automatically stretch it as you add items. You can move or resize it at any time later.

Let's start by adding some items to this top level menu. From MAKEMENU's menu, select Item-Add and enter 'Ledger' in response to the prompt. Don't enter the quotes; those are just to make the string stand out in this manual. You'll see that your menu grows to surround the item name, which is now highlighted as the current (and only) menu item.

If you plan on generating source code for this menu–and that's a good way to get started–now select Item-Name to assign a meaningful Pascal constant name for this menu item. You'll see that MAKEMENU provides a default name of 'miLedger1'. This is the concatenation of 'mi' for menu item, the name of your item, and its numeric item key, which is 1 for the first item. If you can guarantee that this will be the only 'Ledger' item in this menu system (and it will be in this example), it's better to pick a constant name that doesn't include the actual item number. In that way, when you integrate these constant names into your application, later modify your menu system, and the item numbers change, your whole application won't need to change as well. In this case, edit the constant name to read 'miLedger'.

Let's add two more items to this top level menu: Item-Add 'Receivables' (adjust its constant name: Item-Name 'miReceivables') and Item-Add 'System' (adjust its constant name: Item-Name 'miSystem'). Now we have three items on the top level menu, and it has grown to cover almost half the width of the screen. The last item added, 'System', is highlighted as the current item.

In this application, we want the 'System' menu item to appear at the far right hand edge of the screen and we want the top level menu to span the entire width. To achieve this, select Item-Move and use the <Right> arrow key to push 'System' as far right as it will go. The menu will stretch along with it. Before pressing <Enter> to accept this position, move the item back to the left about 10 columns. We want to leave space for a drop-down submenu underneath this item.

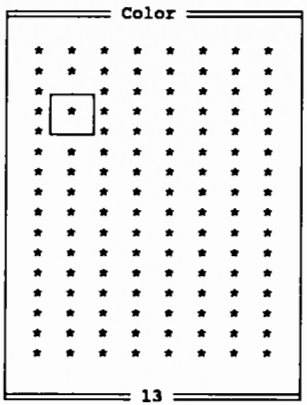
The 'System' item is logically somewhat distinct from the other top level menu items, so let's separate it from them with a span–a vertical bar composed of line drawing characters. To add the span, select Submenu-Settings-Spans-Add. MAKEMENU will select a default set of line-drawing characters (you can change them later if you want) and place them to the right of the current item. Use the <Left> arrow key to move the span into position a few characters to the left of the 'System' item and press <Enter> to accept it.

Let's also take this opportunity to add a header to the top level menu. Select Submenu-Settings-Headers-Add. Another menu will show the positions available for placing the header. Select 'Top left', then enter 'Object Accounting 1.0' in response to the prompt for the header string. The header will appear in the current default color for headers.

OK, the top level menu is in pretty good shape. If we need to, we can return to modify it at any time. For now, let's continue to add a nested submenu to the current item, which should still be 'System'. To do so, select Submenu-Add. An empty submenu will immediately appear below the item. Before we add items to this submenu, however, we should adjust its characteristics.

First, we want the submenu to have a border that merges into the top level menu. To select such a border, choose MAKEMENU's Submenu-Settings-Border command. Another menu showing a comprehensive list of possible borders will appear. Select the one you want (in this case, the eighth item is a good one) and press <Enter>.

Second, we want to use a different color for the unselected items in this submenu. Select Submenu-Settings-Colors-Text to change this particular color. You should see something like this:



Using the cursor keys, move the little box to the middle of the same column. The value at the bottom of the border will be 18 (hex), which corresponds to Dark Gray on Blue. This is the attribute to be used on color systems (note the header at the top). Press <Enter> to accept the new attribute. Now you'll be asked to select the attribute to be used on monochrome systems (note the 'Mono' at the top). This time press <Enter> immediately to accept the default, since there isn't much room for variation on a monochrome display. On a color system, the center of the new submenu will change colors to indicate the new attribute.

Third, we want this drop-down submenu to display its items in a vertical list. Since the global default is horizontal, we'll have to change this submenu by toggling Submenu-Settings-Orientation from 'H' to 'V'.

There are many other adjustments we could make at this point--shadows, explosions, selection characters--but let's continue on to add items to this new submenu.

Select Item-Add and enter 'DOS Shell'. Adjust its constant name by selecting Item-Name and entering 'miDosShell'. Repeat these steps to add the item 'Quit'.

That's enough items for now. Let's test the system a bit. To transfer control from MAKEMENU's menu system to yours, press <F10> (or <ClickBoth> if you're using a mouse). When your menu first gets control, the last item, 'Quit', will be the current item. Try pressing the <Left> arrow three times to move the highlight bar successively to 'Receivables', 'Ledger', and back to 'Quit'.

What?! When you returned to 'Quit' its submenu was no longer pulled down and you got 'System' instead? Not to fear; MAKEMENU has an option that controls this particular behavior. To check it out, first press <F10> to transfer control back to MAKEMENU's menu, then select Defaults-Toggles-Remember pulldown. Now try testing the menu again. You'll see that your menu "remembers" that you had the submenu pulled down.

Only two things remain to be done. First, press <F2> to save the menu system we just created (you can do this while testing your menu system or while within MAKEMENU's menus). Second, go into the main menu and select Object-Write source to generate the source code.

That's all there is to it. You've just finished your first menu system, and we've just finished our little tutorial. There are several other commands and menu choices you need to learn about, of course, but the basic ones are behind you. With a little experimentation, you should be able to figure out how to use the other features of the program on your own after reading the remainder of this section.

The MAKEMENU Menu System

All the capabilities of MAKEMENU are available through its menu system. Many of the commands have hotkeys as well, so that you can save time once you're familiar with the system. What follows is a summary of all the choices in the menu system. If there is a hotkey that corresponds to a given choice, it is displayed to the right of the name of the menu choice.

Note that when an object is loaded, <F10> acts as a hotkey that transfers control between MAKEMENU's menu system and your menu system. <Esc> and <ClickRight> do much the same thing, but they take you to the next highest submenu first; only when the topmost menu is active do they transfer you to the other menu. Also note that, if you have a mouse, <ClickBoth> is equivalent to pressing <F10>.

Defaults

The settings under this item apply to your menu system as a whole. Changing these defaults can have two effects. First, the values you choose will be applied to all further submenus and items that you add during the MAKEMENU session. Second, when it makes sense, MAKEMENU pops up a prompt box "Apply changes globally?" after you've made a change. If you answer "Y", MAKEMENU changes all submenus/items of the menu under construction.

MAKEMENU displays the current default value to the right of many of the items. When you first start MAKEMENU, it uses a set of "factory" defaults. When you read an existing menu system, MAKEMENU adjusts its global defaults to match the settings of the top level menu just read. When you create a new menu system, the defaults retain their previous values.

Backdrop
Lets you select from one of three background types: polka-dot blocks (default), ruler lines, or solid black.

Border
Select a window border. The top portion of the selection list shows a representative example of how the border will appear.

Colors
These menu choices allow you to set various video attributes. The list below shows the name of the menu choice, followed by the names of the corresponding fields in a ColorSet (see the documentation for OPCRT):

```
Text                    TextColor       TextMono
Border                  FrameColor      FrameMono
Header                  HeaderColor     HeaderMono
Shadow                  ShadowColor     ShadowMono
Highlight character     HighItemColor   HighItemMono
Selected item           SelItemColor    SelItemMono
Protected item          ProItemColor    ProItemMono
Mouse cursor            MouseColor      MouseMono
Help line               (stored in Menu object only)
```

Effects

Delay
Prompts for the delay (in milliseconds) between each stage of an explosion.

Explosions
Toggles between exploding and non-exploding windows.

Shadows
Toggles between Big shadows, Small shadows, and no shadows (None).

Sound
Toggles sound effects (during explosions) on and off.

Left select
Prompts for the character to draw to the left of a selected menu item. Displays a full ASCII set of characters; select #0 to draw no character to the left of a selected item.

Right select
Prompts for the character to draw to the right of a selected menu item.

Help

Active

Toggles between displaying a help row or not. Help text for each item is entered with the Item-Help text command. When help is active, MAKEMENU generates source code for a current item procedure that displays the help row.

Row

Prompts for a row where the help line will appear. In MAKEMENU, the entire row is reserved for the help. You can modify the generated source code to display the help in a bordered window or to create other custom effects.

Orientation

Toggles between horizontal and vertical item list orientation.

Source

Color constants

This option setting affects the way source code is generated. If it is On ('Yes'), all video attributes are represented using the constant names defined in OPCOLOR (e.g., 'WhiteOnBlack'), and OPCOLOR appears in the USES list. If it is Off ('No'), hexadecimal numbers (such as $0F) are used instead.

User proc hooks

This option determines whether or not the source code generated includes shells for user-written procedures of the following types: item string customization, error handler, and context-sensitive help.

Case statement

This option determines whether or not the source code generated includes a sample Case statement for handling all of the item keys that may be returned from a call to Menu.Process.

String constants

If this option is enabled, the generated source code declares all of the item name strings and help text strings as typed constants, and makes calls to AddItemPtr (and AddItemHelpPtr) rather than AddItem (and AddItemHelp). The typed constant approach reduces heap memory usage at the expense of source code that is a little harder to read and modify.

Toggles

These menu choices allow you to activate/deactivate various menu options. The list below shows the name of the menu choice, followed by the corresponding menu option (see the OPMENU section earlier in this chapter). Mouse support affects the way MAKEMENU generates source code; it has no real effect on the object itself.

```
Arrow pulldown       mnArrowSelect
Character match      mnAlphaMatch
Character select     mnSelectOnMatch
Erase after select   mnPopOnSelect
Item key help topic  mnUseItemForTopic
Mouse support        mnMouseSupport
Remember pulldown    mnAllowPending
Select on click      mnSelectOnClick
```

Submenu

Add
Attaches a nested submenu to the current item. You can do this only once per item. When you do so, MAKEMENU will draw a small box that shows you where the submenu will appear. (For unbordered submenus, the box will be invisible if you are using the plain backdrop and a black background for menu text.) The submenu will grow automatically as you add items to it, or you can use the Submenu-Resize command to change it manually.

Drag
Repositions the current submenu and any menus nested under it. You won't see the nested submenus move, but the next time you select them, you'll see the difference.

The following commands are available while moving a submenu. The cursor is positioned at the upper left corner of the menu to indicate that the submenu is being moved (rather than resized). Note that, if a mouse is installed, the cursor can also be moved with the mouse.

<Left>, <CtrlS>
Cursor left one column.

<Right>, <CtrlD>
Cursor right one column.

<Up>, <CtrlE>, <CtrlW>
Cursor up one row.

<Down>, <CtrlX>, <CtrlZ>
Cursor down one row.

<Home>, <CtrlQ><S>
Cursor to minimum allowable column.

<End>, <CtrlQ><D>
Cursor to maximum allowable column.

<ShiftTab>, <CtrlLeft>, <CtrlA>
Cursor to previous horizontal tab stop (every 8 columns).

<Tab>, <CtrlRight>, <CtrlF>
Cursor to next horizontal tab stop (every 8 columns).

<PgUp>, <CtrlR>
Cursor to previous vertical tab stop (every 5 rows).

<PgDn>, <CtrlC>
Cursor to next vertical tab stop (every 5 rows).

<CtrlHome>, <CtrlPgUp>, <CtrlQ><E>, <CtrlQ><R>
Cursor to minimum allowable row.

<CtrlEnd>, <CtrlPgDn>, <CtrlQ><C>, <CtrlQ><X>
Cursor to maximum allowable row.

<AltH>
Center window horizontally.

<AltV>
Center window vertically.

<AltC>
Center window both horizontally and vertically.

<Enter>, <ClickLeft>
Accept size/position.

<Esc>, <ClickRight>
Cancel changes (not allowed for Drag).

Move **<AltM>**
Repositions the current submenu without moving its descendants. Available cursor commands are the same as for the Drag command.

Resize **<AltR>**
Resizes the current submenu. The number and length of the current item strings sets a minimum size on the submenu. Available cursor commands are shown above with the Drag command.

Settings
These settings control just the current submenu and its item list. To change settings globally, see Defaults above.

Border
Select a window border.

Colors
These menu choices allow you to set various video attributes. The list of colors is the same as for Defaults-Colors above.

Effects
These menu choices allow you to change various special effects. With one addition, the list is the same as for Defaults-Effects above. The addition is Padding, which prompts for the number of blanks used to pad each item name.

Headers

Add
Displays a submenu showing the 6 possible positions for the header. Once the position is selected you will be prompted for the header itself, and then given a chance to set the colors it is to be displayed in.

Edit
Displays a pick list of existing headers. Move the cursor to the header you want to edit and press <Enter>. You can change both the text of the header and its colors.

Remove
Displays a pick list of existing headers. Move the cursor to the header you want to remove and press <Enter>.

Orientation
Changes the orientation of the current submenu. Note that the window borders will be resized to surround the items.

Spans

Add

Computes the span characters to match the current submenu border, and places a default span after the current menu item. Use the arrow keys to position the span as desired and press <Enter>.

Edit

Displays a pick list of existing spans. Move the cursor to the span you want to edit and press <Enter>. Use the arrows keys to reposition the span and press <Enter>. Select a new character set for the span from a pick list and new video attributes from a color chart.

Remove

Displays a pick list of existing spans. Move the cursor to the span you want to remove and press <Enter>.

Kill

Disposes of the current submenu, its items, and all submenus nested within it. You'll be given an opportunity to confirm this action.

Item

Add

Prompts for the name of an item to add to the current submenu. You must add each item before you can edit, move, or perform any other action on it. The new item is added just after the currently selected item.

Name

Prompts for the symbolic name to be used to refer to this menu item. The name must be a valid Pascal identifier (MAKEMENU checks this), and it must be unique–no two items should have the same name (MAKEMENU does *not* check for this). Entering a name is optional. If you do not specify a name, the source code generated by MAKEMENU will refer to the item as 'miXxxxxNnn' where 'Xxxxx' is up to 10 valid characters of the item's displayed string and 'Nnn' is the item's key number.

Edit

Edit the displayed string for the current item.

Key

Modify the key of the current item. MAKEMENU assigns keys sequentially as items are added and will renumber the items on request (see Object-Rekey). The number to the right of Key is the key of the current menu item.

Select char

Choose a different selection character for the current item. Use the cursor keys or the mouse to reposition the character highlight. Move the cursor all the way to the left to disable item selection by character.

Help text

Assign help text to the current item. MAKEMENU will prompt for new text or allow you to edit existing help. The help text will be displayed only if you enable Defaults-Help-Active.

Move
Adjust the position of the current item. Use the arrow keys and press <Enter> to accept the change. The submenu borders will be adjusted if the item "pushes" against them. The position of any submenu attached to the current item is not affected.

Note that Move does not change the order of the linked list of items, so it is a mistake to move an item past the items that surround it. If the intent is to do so permanently, use the Reorder command instead.

Reorder
Adjust the relative position of the current item in the linked list of items.

Drag
Adjust the position of all items simultaneously.

Kill
Delete the current item. If it has submenus, you'll be prompted whether to proceed, since all nested submenus will be deleted as well.

Object

Load
Load an existing menu system from the current library or create a new one.

Save **<F2>**
Save the current menu system.

Rename
Change the name of the current menu system.

Save as
Save the current menu system under a new name.

Info **<AltI>**
Display information about the object being created. Note that, if no object is loaded, <AltI> will display information about the current library instead (see Library-Info, below).

Test **<F10>**
Test the menu system being designed.

Write source
Generate source code for the current menu system.

Document
Generate documentation for the current menu system.

Rekey
Assign new key numbers to all menu items, starting with 1 and increasing sequentially in left-to-right, depth-first order.

Library

Load **<F3>**
Load a new or existing library. If the file specified does not exist, you will be asked if you want to create it.

New
Create a new library. If the file specified already exists, you will be asked if you want to overwrite it.

Delete
Displays a list of all objects in the library. Press <Enter> to mark an object for deletion (a checkmark will appear beside the name) or to remove an existing deletion mark. Press <Esc> when you are finished marking object names. Note that the objects will not be physically removed from the library until it is packed.

Info **<AltI>**
Display information about the current library. Note that <AltI> will execute this command only if no object is loaded; otherwise, it will display information about the current object (see Object-Info).

Pack
Pack the current library if packing is needed. Packing purges the library of earlier versions of objects, as well as objects marked for deletion.

Quit **<AltX>**
Quit the program.

Generating Source Code

As we've seen, there's not much involved in generating source code. You simply select the appropriate menu choice and enter the name of the source file to be generated. If the file specified already exists, you will be asked if it should be overwritten. Be careful here. You don't want to overwrite a "real" program of the same name. For this reason, it is generally a wise practice to give your menu a different name than the program it will ultimately be used in, especially if all your files are in the same directory.

We should reiterate that there are six menu choices that affect the source code that is produced by MAKEMENU: "Color constants," "User proc hooks," "Case statement," "String constants," (Defaults-Source) "Mouse support," (Defaults-Toggles) and "Help-Active" (Defaults-Help). With the exception of String constants, these choices, explained earlier, all correspond to options that are On by default.

Generating Documentation

In addition to generating source code, MAKEMENU can also generate "documentation": a brief report showing information about the menu system and the items that are in it. If you were to generate documentation for the sample entry screen MENU1 (in SAMPLE.OPL), the report would look something like this:

```
Menu System Documentation
-------------------------
Object:                 MENU1
Library:                SAMPLE.OPL
Size when last saved:   1213

Menu items:             7
Highest item key:       7
Submenus:               3
Item name                                X   Y   Const          Number
---------------------------------------  --- --- -------------  ------
' File '                                 5   2   miFile          = 1
    'Load'                               4   5   miLoad          = 2
    'Store'                              4   6   miStore         = 3
    'Quit'                               4   8   miQuit          = 4
' Options '                             19  2   miOptions       = 5
    'Colors'                             18  5   miColors        = 6
    'Directories'                        18  6   miDirectories   = 7
```

The first section of the report shows information about the menu system as an object: its name, the library it's in, and its size.

The second section shows the number of items in the menu, the highest key number (often the same as the number of items, but not always), and the number of submenus (counting the top level menu).

The last section is a list of the items in the menu system. For each item, the documentation shows its name string, the coordinates where it will be displayed, the Pascal constant name for the item key, and the key value. The indentation of the name string indicates the submenu grouping and hierarchy: names at the same level of indentation are in the same submenu; the submenu's parent is the next item above with reduced indentation.

Normally, the report is sent to a text file with an extension of "DOC". If you want it sent to the printer instead, enter "PRN." (or "LPT1." or "LPT2.") when prompted for a filename.

Using a MAKEMENU Library Directly

If an application uses several menu systems or a single very large menu system, it may be to your advantage to load the menus directly from a library instead of having MAKEMENU generate source code that you include in your program. Whether this cuts memory usage or not will depend on the size of your menus. When you load from a library, the compiler doesn't need to link in some of the methods used to build menus from source statements, e.g., AddItem and AddSubMenu. On the other hand, the compiler must link in methods for opening and reading libraries. When the menus get big enough, the scales tip in favor of the library approach because there is no need for repeated calls to AddItem and AddSubMenu, nor are there many literal strings that get duplicated on the heap.

It's easy for your program to read a library that MAKEMENU has created. The following example program shows how:

```
program LoadMenuFromLibrary;
uses
  Dos, OpRoot, OpCrt, OpFrame, OpWindow, OpMenu;
var
  M : Menu;
  Status : Word;
```

```
function InitMenuFromLib(LibName : PathStr;
                         ObjName : DirEntryName;
                         var M : Menu) : Word;
  {-Initialize a menu from a library}
var
  Lib : Library;
  Code, Version, Status : Word;
begin
  {Open the library and register the types needed for menus}
  if not Lib.Init(LibName, SOpen, 4096, '') then begin
    InitMenuFromLib := InitStatus;
    Exit;
  end;

  Lib.RegisterHier(MenuStream);
  Lib.RegisterHier(ExplodingWindowStream);

  {Validate and load the specified menu system}
  Lib.GetCodeAndVersion(ObjName, Code, Version);
  if Code <> otMenu then begin
    {Return code indicating an unexpected object type}
    InitMenuFromLib := epFatal+ecIdNotRegistered;
    Exit;
  end;
  Lib.GetEntry(ObjName, M);
  InitMenuFromLib := Lib.GetStatus;

  {Close the library}
  Lib.Done;
end;

begin
  {Load the menu system}
  Status := InitMenuFromLib('SAMPLE.OPL', 'MENU1', M);
  if Status <> 0 then begin
    WriteLn('Unable to initialize menu, Status = ', Status);
    Halt(1);
  end;

  {Try the menu system}
  M.Process;
  M.Erase;
  M.Done;
end.
```

The example opens the specified library file ('SAMPLE.OPL' in this case, so you can try running it if you like). It specifies a 4K buffer to speed reading of the library. Since MAKEMENU doesn't write a custom signature to its library files, the last parameter to Lib.Init must be an empty string. Next the example registers the data types for the Menu object. If your menu systems don't use exploding windows, you don't need to register the ExplodingWindowStream hierarchy.

GetCodeAndVersion returns codes that indicate the type of the specified library entry. If the menu object name is hard-coded into a real application, you can skip this validation step. GetEntry reads the library file to instantiate the Menu object M. GetEntry triggers a series of calls to Menu.Load and to the Load constructors of other related objects. Once the Menu object is initialized, the library file is closed to regain both a file handle and the associated heap space.

Finally, the example demonstrates that the menu system was loaded successfully by calling its Process method.

If your application requires special features that MAKEMENU can't store in the library, it's possible to activate them after you've loaded the menu. After the call to InitMenuFromLib, you would insert calls to other Menu methods–EnableAllAlternateOpen, SetCurrentItemProc, SetCustomStringProc, ProtectItem, DefaultItem–just the same as if the menu had been initialized by source statements.

Limitations

MAKEMENU lets you design menu systems with almost every feature that OPMENU itself supports. There are a few things you can't do with MAKEMENU, however. Most of the missing capabilities are omitted because the characteristics they describe cannot be known until the surrounding application is in place; these include

- string customization routine
- generalized help row display (in a bordered window, for example)
- protected items
- non-standard default items
- custom alternate open routines

One other capability–multi-character left and right selection strings–is left out because supporting it would make entry of the characters much more difficult for you. As it is, MAKEMENU provides a full ASCII character selection list so you can pick arrows and other control characters commonly used for selection markers.

2. PSCREEN

PSCREEN is a utility program designed to be used in conjunction with the OPWINDOW unit. It operates in either of two modes. In resident mode, PSCREEN allows you to capture text screens, video attributes and all, in a packed format and save them to disk. In non-resident mode, PSCREEN allows you to display a packed window that has previously been saved to a file. This facility can be used to create colorful batch files, for example. The PackedWindow object in OPWINDOW can also read and display the contents of the packed window files within an application program.

Using PSCREEN

To run PSCREEN in resident mode, simply enter

```
PSCREEN
```

at the DOS prompt, with no parameters. PSCREEN is a swappable TSR (see Chapter 11) which allocates only 6K of DOS memory; the rest of the program is stored in expanded (EMS) memory or on disk. Optionally, you can enter any of the following parameters after PSCREEN:

```
/E          don't use EMS
/M          disable swapping Messages
/N          No swapping
/Ppathname  specify Pathname to use for swapping
/U          Unload PSCREEN from memory
/?          display this help screen
```

/E prevents PSCREEN from allocating EMS memory for swapping. Otherwise, it will allocate 48K of EMS, if available.

/M disables the 'Swapping in' and 'Swapping out' messages that PSCREEN normally writes when it is swapping to disk. /M is especially useful if you're swapping to a RAM disk, which is so fast that the messages are a nuisance.

/N disables swapping altogether. PSCREEN will consume about 52K of DOS RAM in this case.

/P allows you to specify an alternate location for PSCREEN's swap files. There are two of them, each about 45K bytes. By default, the swap files are stored in the root directory of drive C. Specify just the drive and directory, immediately following the /P option. The swap files are always named 'PSCREEN1.$$$' and 'PSCREEN2.$$$', but they are marked hidden, so you may never see them.

/U unloads PSCREEN from memory. Like other **Object Professional** TSR's, PSCREEN will unload only when it is safe to do so.

/? writes a brief help screen listing these same options.

When you want to save a packed window to disk, press the PSCREEN hotkey, <Alt><LeftShift><P>. The cursor will then change to a blinking block.

To select the region of the screen (that is, the 'window') that you want to save, move the cursor to the top left corner of the region and press <Enter>, then move the cursor to the bottom right corner and press <Enter> again. As you move the cursor around, the currently selected portion of the screen will be repainted to indicate the bounds of the window. If you want to cancel the operation, press <Esc>. If you want to unload PSCREEN from memory, press <AltU> instead.

Once you've selected a window, you'll be asked to enter the name of the file to save it in. Again, you can unload PSCREEN by pressing <AltU> when prompted for a filename, or cancel the operation by pressing <Esc>.

To display a packed screen using PSCREEN, enter

```
PSCREEN filename.ext
```

at the DOS prompt. PSCREEN will simply display the contents of the file, wait for a keypress, and then terminate. It will not attempt to go resident.

Displaying a Packed Window File in a Program

It's easy to display a file that PSCREEN has created. Here's a brief example, where FName is the name of a packed window file.

```
program DisplayPackedWindow;
uses  OpCrt, OpWindow;
const FName = 'TEST.PWF';
var   PW : PackedWindow;
begin
  if PW.Read(FName) then
    PW.Display
  else
    WriteLn('Error ', InitStatus, ' reading ', FName);
end.
```

It's also possible to bind the packed window file into the application's EXE file. See PackedWindow.InitFromMemory in section C of this chapter for an example.

3. WINWOW

WINWOW doesn't do anything useful, unless perhaps it gives you something to meditate upon while you're trying to put off writing some boring documentation. We've also found WINWOW useful as a test program for alternate (higher resolution) text modes supported by EGA and VGA cards. WINWOW adjusts itself to utilize however many rows and columns are available when it starts up.

WINWOW demonstrates some of the visually impressive features of the OPWINDOW unit. It starts by creating 8 popup windows of random sizes and colors, then begins moving them around the screen at a frenzied pace–scrolling, expanding, and contracting them as they go. Press any key to regain control of the helm and return to DOS.

Here's a list of the **Object Professional** features that WINWOW shows how to use:

- main and alternate window frames
- adding shadows and headers
- selecting non-default video attributes for a window
- enabling window explosions
- creating a virtual screen and writing to it
- copying from a virtual screen to a window to achieve scrolling effects
- window moving and resizing
- random-access window stacking
- erasing all windows from the stack

5. Text Editors and File Browsers

A. Overview

The units described in this chapter provide objects that can be used for editing large blocks of text and for browsing through files. Each of these objects is derived, either directly or indirectly, from the CommandWindow object described in the documentation for OPWINDOW.

The first of these units, OPMEMO, implements an object called a Memo that allows "basic" text editing of the sort required for memo fields in database records. OPMEMO also implements a derivative of the Memo, called a MemoFile, that allows the same kind of editing to be performed on data stored in a text file.

The OPEDITOR unit implements a derivative of the MemoFile, called a TextEditor, that provides "advanced" text editing capabilities: numerous block-oriented commands, search-and-replace, text markers (sometimes referred to as "bookmarks"), "smart tabs" and auto-indent mode. A TextEditor can do almost everything the Turbo Pascal editor does, and quite a few things that it can't.

OPBROWSE provides a facility, in the form of the Browser object, for "browsing" through files of any kind, binary files as well as text files. Although a Browser does not provide any means for modifying the data it displays, it has one important advantage over the TextEditor: it works with files of any size.

B. OPMEMO: Basic Text Editor

This section is divided into three parts. The first covers the lowest-level text editing object in **Object Professional**, the Memo. The second covers one of its two derivatives, the MemoFile. And the third covers the procedures and functions that are interfaced by OPMEMO.

1. The Memo

The Memo object gets its name from the term "memo field," commonly used by database programmers to describe a field used to store text entered by a user in a free-form fashion. Consider, for example, a simple database used to store names and addresses: a "phone book." Most of the fields in each database record would contain information needed for each person in the database: last name, first name, address, city, state, zip code, phone number. For the phone book to be truly useful, however, it must also provide the user a way to store information that may not be needed for each person: birthday, wedding anniversary, spouse's name, children's names, as well as information that can't really be categorized.

If you've looked at the phone book implemented in DESKPOP, you know that it meets the need by providing a Notes field. You may also have noticed that it imposes a rather severe limit on the amount of data that can be stored in that field: all the data has to fit in 4 lines of no more than 40 characters each. The limit exists because each Note is stored in an 'array[1..4] of string[40]' and is edited using OPFIELD's multi-line editor. Although we could increase the limit by adjusting the maximum number of lines and their maximum length, we would soon realize that the method of storage was terribly inefficient, since blank lines require as much space as full ones. If we wanted to increase the limit substantially, we would need to switch to a more efficient means of storage, one that allowed a variable number of lines of different lengths to be stored in memory one after the other, in an 'array[1..n] of Char', and a more flexible editor that could take advantage of it. This is the type of situation that the Memo object was designed to handle.

Specifically, the Memo editor was designed to work with data stored in a text buffer of a fixed size (< = 65,521 bytes). Each line in the buffer is delimited by a carriage return/line feed combination (^M^J), and the last line in the buffer is followed immediately by a ^Z character. Although this format is identical to that of a Turbo Pascal text file, it works equally well for memos that are stored as portions of some other binary data.

Now that you have some idea of what a Memo object is, we can start examining the question of how to use one. We'll begin by looking at a simple example program, shown below.

```
program MemoExample; {EXMEMO.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const
  BufSize = 5000;
var
  M : Memo;
  Buffer : array[1..BufSize] of Char;
  ExitCommand : Word;
begin
  {mark the text buffer as empty}
  Buffer[1] := ^Z;
```

```
  {initialize memo editor with custom window options}
  if not M.InitCustom(2, 4, 79, 24,                    {Window coordinates}
                      DefaultColorSet,                 {ColorSet}
                      DefWindowOptions or wBordered,   {Window options}
                      BufSize,                         {Buffer size}
                      @Buffer)                         {Buffer pointer}
  then begin
    WriteLn('Failed to init Memo. Status = ', InitStatus);
    Halt;
  end;

  {Use built-in status and error handlers}
  M.SetStatusProc(MemoStatus);
  M.SetErrorProc(MemoError);

  repeat
    M.Process;
    ExitCommand := M.GetLastCommand;
    case ExitCommand of
      ccQuit, ccError : {do nothing} ;
      {...process user exit commands...}
    end;
  until (ExitCommand = ccQuit) or (ExitCommand = ccError);

  M.Erase;
  M.Done;
  ClrScr;
end.
```

The first order of business here is to set up the text buffer to be used by the Memo editor. In this case, the buffer is a static array 5000 (BufSize) bytes in size. We could allocate it on the heap if we wanted to, but for the purposes of demonstration we've kept things simple. Since we don't have any existing data to edit in this case, we mark the buffer as empty by putting a ^Z character in the first element of the array.

The next task is to initialize the Memo editor. Even though we're using the default ColorSet provided by OPCRT, we have to use the InitCustom constructor because we want the edit window to be bordered. Besides the parameters common to all InitCustom constructors for window-based objects (window coordinates, colors, and window options), Memo.InitCustom requires two additional parameters. The first indicates the size of the text buffer to use, and the second its address.

In looking at the call to InitCustom, you may have noticed that the second parameter calls for the top row of the edit window to appear at row 4, placing the frame on row 3. This reserves the top two rows of the screen for other things, in this case a status line (row 2) and a line for error messages (row 1). The status line–which shows, among other things, the current location of the cursor–is displayed by a built-in routine called MemoStatus, enabled by calling the SetStatusProc method. Error messages are displayed by another built-in routine, MemoError, enabled by calling the SetErrorProc method, inherited by Memo from CommandWindow.

Now we're ready to start editing. The main program loop just does two things over and over: it calls the Process method to display the edit window and process editing commands, and it calls GetLastCommand to determine the command used to exit from the editor. In this simple case, only two exit commands are likely–ccQuit, which means quit editing, and ccError, which means that a fatal error occurred while editing–and the loop ends when either of them is returned.

The program ends with some basic "cleanup code." The Memo window is erased and destroyed, and the entire screen is cleared to get rid of any text on the top two rows of the screen.

Although this is a somewhat atypical example, it does show the basic steps involved in setting up a Memo window and editing text with it. A more typical example would probably be one in which the text buffer is part of a more complex data structure, likely a record variable of some sort. Despite being written primarily as a demonstration of OPENTRY's capabilities, ENTRY.PAS is in this respect an equally appropriate example of how to use OPMEMO.

The sections that follow describe the finer points of using OPMEMO. Since the most important of these is the use of the editor itself, we'll start with a list of the editing commands it provides.

Editing Commands

OPMEMO offers a fairly complete set of basic text editing commands, arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant MemoKeySet.

Cursor Movement

ccLeft **<Left>, <CtrlS>**
Cursor left one character. If the cursor is at the beginning of a line, and the meWrapAtLeft option is in effect, it is moved to the end of the previous line. If the option is not in effect, the cursor does not move.

ccRight **<Right>, <CtrlD>**
Cursor right one character.

ccWordLeft **<CtrlLeft>, <CtrlA>**
Cursor left one word. If the cursor is at the beginning of a line, it is moved to the end of the previous line.

ccWordRight **<CtrlRight>, <CtrlF>**
Cursor right one word. If the cursor is at the end of a line, it is moved to the beginning of the following line.

ccHome **<Home>, <CtrlQ><S>**
Cursor to beginning of line.

ccEnd **<End>, <CtrlQ><D>**
Cursor to end of line.

ccUp **<Up>, <CtrlE>**
Cursor up one line.

ccDown **<Down>, <CtrlX>**
Cursor down one line.

ccScrollUp **<CtrlW>**
Scroll display up one line.

ccScrollDown **<CtrlZ>**
Scroll display down one line.

ccPageUp **<PgUp>, <CtrlR>**
Scroll display up one page.

ccPageDn **<PgDn>, <CtrlC>**
Scroll display down one page.

ccScreenTop **<CtrlHome>, <CtrlQ><E>**
Move cursor to top of edit window.

ccScreenBot **<CtrlEnd>, <CtrlQ><X>**
Move cursor to bottom of edit window.

ccTopOfFile **<CtrlPgUp>, <CtrlQ><R>**
Move cursor to beginning of file.

ccEndOfFile **<CtrlPgDn>, <CtrlQ><C>**
Move cursor to end of file.

Insertion and Deletion

ccDel **<Del>, <CtrlG>**
Delete character at cursor.

ccBack **<Bksp>, <CtrlH>, <CtrlBksp>**
Delete character to left of cursor. If the cursor is at the beginning of a line, the line will be joined with the previous line.

ccDelLine **<CtrlY>**
Delete current line.

ccDelEol **<CtrlQ><Y>**
Delete from cursor to end of line.

ccDelWord **<CtrlT>**
Delete word to right of cursor. If the cursor is at end of a line, the following line is joined with the current line.

ccSelect **<Enter>, <CtrlM>**
Start a new line.

ccInsertLine **<CtrlN>**
Insert a new line at the position of the cursor.

ccTab **<Tab>, <CtrlI>**
Move the cursor to the next tab stop. If insert mode is on, any text to the right of the cursor is moved to the right of the tab stop.

ccCtrlChar **<CtrlP>**
Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is not available if the video attribute for control characters is the same as that for ordinary text and the teMapCtrls option is on.

Mode Toggles

ccIns **<Ins>, <CtrlV>**
Toggle insert mode on and off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

ccIndent **<CtrlO><I>**
Toggle auto-indent mode on or off. In auto-indent mode, pressing <Enter> while in insert mode will cause the new line inserted to have the same indentation level as the previous line. Auto-indent mode also affects the way that text is formatted when word wrap occurs--the new line will have the same indentation level as the previous line--and hence the behavior of the reformatting commands (<CtrlB> and <CtrlK><G>).

ccWordWrap **<CtrlO><W>**
Toggle word wrap on and off. When word wrap is on, any attempt to insert or append text beyond the right margin will cause a new line to be inserted following the current line and all words that are at least partially beyond the right margin to be moved to the new line.

Text Formatting

ccReformatP **<CtrlB>**
Reformat the current paragraph. Note that this command does nothing if word wrap is off.

ccReformatG **<CtrlK><G>**
Reformat the entire text buffer (Global reformat). Note that this command does nothing if word wrap is off. Use this command with caution!

Other

ccRestore **<CtrlQ><L>**
Restore original contents of the current line.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling MemoCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a help procedure, the Memo will pass it the following three parameters: ucMemo, @Self, and the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

ccQuit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit editing.

ccMouseSel **<ClickLeft>**
Moves the cursor to the position indicated by the mouse. This command can also be used to scroll the edit window by clicking on a scroll bar.

In addition to the commands shown above and the standard exit commands (ccError and the user exit commands--ccUser0, etc.), Process also recognizes two other special commands, ccNewFile and ccSaveFile, that are not assigned to any keys in the default command table. This simply provides a hook for programs that use the MemoFile object for editing text files. The commands, which do nothing, are considered to be exit commands, and it is up to the programmer to implement them if desired.

Memo Field Editing

As indicated earlier, the most common use for the Memo object is for editing memo fields in a database application. When using it in this fashion, there are several special issues that you need to keep in mind.

Most importantly, the Process method assumes that the ReinitBuffer method has been called beforehand to initialize various internal variables that maintain information about the contents of the text buffer--the number of characters in it, the number of lines in it, etc. Memo's

constructors make this call for you the first time around–that's why our earlier example program did not contain a call to ReinitBuffer. But in many applications a single Memo will be instantiated once at the beginning of the program and reused many times. In such cases, the contents of the text buffer might very well have changed from one Process call to the next, making it imperative to call ReinitBuffer, like so:

```
M.ReinitBuffer;
repeat
  M.Process;
  ExitCommand := M.GetLastCommand;
  case ExitCommand of
    ccQuit, ccError : {do nothing} ;
    {...process user exit commands...}
  end;
until (ExitCommand = ccQuit) or (ExitCommand = ccError);
```

This is how the main loop of our sample program would have looked if it were being executed repeatedly. For a good example of a case like the one described above, see ENTRY.PAS.

Two things that people commonly want to do with memo fields is limit the number of lines that may be entered by the user and/or limit the maximum number of characters in a given line. To meet this need, Memo provides the SetLineLimit and SetMaxLength methods. See the entries for these two methods for further details.

One slightly less common need is to use a Memo for display purposes only, allowing the user to examine the contents of a memo field but not to modify it. Memo thus provides the meReadOnly option, which temporarily disables all editing commands that might alter the contents of the text buffer. (The list of all such commands is stored in the global typed constant DisallowedInReadOnlyMode.) This option corresponds to OPENTRY's esReadOnly option for EntryScreens.

Constants

```
BadMemoOptions : LongInt = meModified+meReformatting;
```

Options that exist for internal use, and may not be altered by calling meOptionsOn or meOptionsOff.

```
DisallowedInReadOnlyMode : set of Byte =
 [ccChar, ccCtrlChar, ccSelect, ccInsertLine, ccBack, ccDel, ccRestore,
  ccDelEol, ccDelLine, ccDelWord, ccIns, ccTab, ccIndent, ccWordWrap,
  ccReformatP, ccReformatG];
```

The commands in this set are disallowed by the memo editor when running in read-only mode (using the meReadOnly option) because they would modify the text being displayed.

```
DefMemoOptions : LongInt = meInsert+meIndent+meWordWrap+meMapCtrls+
                           meMousePage+meMakeBackups;
```

The default options for a Memo or MemoFile.

```
DefTabDelta : Byte = 8;
```

The default distance between tab stops.

```
DefMaxLength : Byte = 254;
```

The default maximum length for a line of text.

```
meInsert        = $00000001;  {True if in insert mode}
meIndent        = $00000002;  {True if in auto-indent mode}
meReadOnly      = $00000004;  {True if in read-only mode}
meWordWrap      = $00000008;  {True if word wrap is on}
meDeleteJoins   = $00000010;  {True if <Del> can join lines}
meModified      = $00000020;  {True if edits have been made; internal use}
```

```
  meIndentIsPara   = $00000040;  {indent starts paragraph?}
  meMousePage      = $00000080;  {clicking on scroll bar scrolls by page}
  meAllowTrunc     = $00000100;  {read partial files?}
  meMapCtrls       = $00000200;  {map control characters?}
  meMakeBackups    = $00000400;  {make backup files?}
  meStreamReload   = $00000800;  {reload previous file when activating a
                                  MemoFile stored in a stream}
  meReformatting   = $00001000;  {flag set while reformatting; internal use}
  meNewFile        = $00002000;  {new file loaded}
  meDeallocate     = $00004000;  {deallocate the buffer when Done is called?}
  meWrapAtLeft     = $00008000;  {wrap to prev line on <Left> at column 1}
  meNoCtrlZ        = $00010000;  {suppress ^Z when writing to disk}
```

These are the options that affect the behavior of a Memo or MemoFile. meInsert selects insert mode, in which new text is inserted at the position of the cursor rather than overwriting existing text. meIndent selects auto-indent mode, in which pressing <Enter> will cause a new line to be inserted that has the same indentation level as the previous line. meReadOnly selects read-only mode, in which the text in the text buffer may be examined but not modified. meWordWrap enables word wrap; when word wrap is on, any attempt to insert or append text beyond the right margin will cause a new line to be inserted following the current line and all words that are at least partially beyond the right margin to be moved to the new line. If meDeleteJoins is in effect, pressing <Del> at the end of a line causes the following line to be joined with the current line at the position of the cursor.

meModified is an internal flag that is set whenever the text buffer is modified. Note that if the user undoes his changes to a given line before leaving that line, the modified flag will be restored to the state it was in before the changes were made. meIndentIsPara affects the behavior of the reformatting commands. If this option is off, blank lines are assumed to mark the end of one paragraph and the start of another. If it is on, blank spaces at the beginning of a line can also signal the start of a new paragraph. meMousePage determines what happens when the user clicks on the up and down arrows of a scroll bar. If the option is off, the edit window will be scrolled up or down by a single line; if it is on, the edit window will be scrolled up or down one full page.

meAllowTrunc affects the behavior of MemoFile.ReadFile. If this option is on, and ReadFile is asked to load a file that is too large to fit within the text buffer, it will read as much of the file as it can and generate a warning (epWarning+ecFileTruncated). If it is off, it will not attempt to read the file, and it will generate a fatal error (epFatal+ecFileTooLarge). meMapCtrls determines how control characters other than ^M, ^J, and ^Z (always suppressed) are displayed when they appear in the text buffer. If it is on, control characters are displayed as uppercase alphabetics (^A -> 'A') in an alternate color; if it is off, they are displayed normally. meMakeBackups affects the behavior of MemoFile.SaveFile. If it is on, and the file being saved already exists, the existing file will be renamed (given an extension of BAK) before the new version of the file is saved. If it is off, the existing file will be overwritten.

meStreamReload affects the behavior of MemoFile.Load and MemoFile.Store. If it is on, Store saves the name of the file currently being edited in the stream. The next time that Load is called, it will attempt to reload the file whose name was saved. meReformatting is an internal flag that is set while a paragraph or file is being reformatted. It is used to signal that screen updates and error messages should be temporarily suppressed. meNewFile is a flag set by MemoFile.ReadFile when a new file has been loaded into the text buffer. Its purpose is to allow status routines that display the name of the current file to determine when the file has changed; after the status line has been updated, the flag should be cleared.

meDeallocate is a flag intended primarily for internal use. It is set by any MemoFile constructor that allocates a text buffer, and signals that the destructor (Done) should deallocate the buffer. You may set this flag yourself in cases where you have allocated your own buffer for a Memo or MemoFile and want Done to deallocate it for you. meWrapAtLeft determines what happens when <Left> (ccLeft) is pressed when the cursor is at column 1. If this option is on, the cursor will "wrap" to the end of the previous line; otherwise it will remain where it is. meNoCtrlZ affects the behavior of MemoFile.SaveFile. If it is off, as it is

by default, the ^Z that marks the end of the text buffer will be written to disk; if it is on, the ^Z will not be written.

```
MemoKeyMax = 250;
MemoKeyID  : string[16] = 'opmemo key array';
MemoKeySet : array[0..MemoKeyMax] of Byte = (...);
MemoCfgEnd : Byte = 0;
```

MemoKeyId is an ID string used to mark the beginning of the configuration data area for OPMEMO; MemoCfgEnd marks the end of the configuration data area. In between them is MemoKeySet, the command table used by MemoCommands. MemoKeyMax is the last valid index into the table.

```
meWordDelims : CharSet = [^I, ' ','"', '.', ',', ':', ';', '?', '!',
                          '*','(', ')', '[', ']', '{', '}', '<', '>',
                          '+', '-', '/', '\', '''', '$', '=', '^', '#'];
```

This set defines the group of characters that are to be treated as word delimiters by the various word-oriented commands (ccWordLeft, ccWordRight, ccDelWord). At the very least, this set should contain ' '.

Types

```
MemoPtr = ^Memo;
Memo =
  object(CommandWindow)
    ...
    meBufSize : Word;         {size of the text buffer}
    meMaxLines : Integer;     {maximum number of lines that may be entered}
    meTabDelta : Byte;        {distance between tab stops}
    meMargin : Byte;          {right margin for word wrap}
    meTotalBytes : Word;      {actual number bytes currently in buffer}
    meTotalLines : Integer;   {actual number of lines currently in buffer}
    meCurLine : Integer;      {line number of current line}
    meCurCol : Byte;          {position of cursor within current line}
    ...
  end;
```

A memo editor. The data fields shown above should not be modified directly. They are documented here primarily to assist in the writing of status routines. Note that Memo maintains a "safety margin" of 2 bytes, meaning that the user is never allowed to enter more than meBufSize-2 bytes of text. (This helps avoid buffer overflow problems in cases where a line must be split because it is too long.) When calculating used/unused space in the text buffer, you should therefore base your calculations on meTotalBytes and meBufSize-2.

```
MemoBufPtr = ^MemoBuf;
MemoBuf = array[1..65521] of Char;
```

A buffer used to store text being edited. Note that 65521 is the *maximum* size of a text buffer; the actual size varies.

```
MemoStatusProc = procedure (MP : MemoPtr);
```

A user-written routine that displays status information (current line, column, etc.). MP is a pointer to the Memo/MemoFile (or TextEditor) that is making the call.

Variables

```
MemoCommands : CommandProcessor;
```

The default CommandProcessor for a Memo or MemoFile.

Done / Init

Declaration

```
destructor Done; virtual;
```

Purpose

Deallocate buffers.

Description

This destructor deallocates the text buffer, if appropriate, and then calls the parent's destructor to deallocate memory used for window buffers, shadows, etc.

Note that, in the case of a MemoFile or TextEditor, Done does not save the contents of the buffer before disposing of it. It is the programmer's responsibility to check the meModified flag and call SaveFile, if appropriate, before calling Done.

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a memo window using default window options.

Description

This constructor instantiates a Memo using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
Buffer[1] := ^Z;
if not M.Init(2, 4, 79, 24, BufSize, @Buffer) then begin
  WriteLn('Failed to init Memo. Status = ', InitStatus);
  Halt;
end;
```

Instantiates M using the default window options and colors. Putting a ^Z in the first position of the buffer marks it as empty.

See Also

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a memo window with custom window options.

Description

This constructor instantiates a Memo using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window, where the text being edited will be displayed. BufferSize is the size of the text buffer to use, and BufPtr is its address. If the buffer is empty at the time InitCustom is called, there should be a ^Z in the first position.

InitCustom will fail only if the parent constructor (CommandWindow.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Only two sets of attributes in the specified ColorSet have special relevance for window objects of type Memo (or MemoFile). TextColor/TextMono determine the attribute used to display ordinary text, and CtrlColor/CtrlMono determine the attribute used to display control characters embedded in the text. After the object has been instantiated, these attributes may be changed by calling the SetTextAttr and SetCtrlAttr methods, respectively.

Example

```
Buffer[1] := ^Z;
if not M.InitCustom(2, 4, 79, 24,                {Window coordinates}
                    DefaultColorSet,             {ColorSet}
                    DefWindowOptions or wBordered, {Window options}
                    BufSize,                     {Buffer size}
                    @Buffer)                     {Buffer pointer}
then begin
  WriteLn('Failed to init Memo. Status = ', InitStatus);
  Halt;
end;
```

Instantiates M as a bordered window that uses the default colors. Putting a ^Z in the first position of the buffer marks it as empty.

See Also

Init
RawWindow.SetTextAttr
SetCtrlAttr

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a memo from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Memo.Store.

If the text buffer was registered before Store was called, and it is registered again prior to the call to Load, then Load will initialize the Memo to use the buffer whose address was registered. It is imperative that the new buffer be the same size as the original, and it is the programmer's responsibility to insure that it is.

If the buffer was not registered in both cases, Load will do one of two things, depending on the state of the meDeallocate flag. If the flag is set, Load will attempt to allocate a buffer on the heap; if it cannot do so, Load will fail and InitStatus will be set to epFatal+ecOutOfMemory. If the flag is not set, Load will Fail immediately and InitStatus will be set to epFatal+ecIdNotRegistered.

The stream registration procedure for a Memo is MemoStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const
  BufSize = 5000;
var
  M : Memo;
  S : BufIdStream;
  Buffer : array[1..BufSize] of Char;
begin
  Buffer[1] := ^Z;
  S.Init('MEMO.STM', SOpen, 1024);
  S.RegisterHier(MemoStream);
  S.RegisterPointer(1000, @Buffer);
  S.Get(M);
  if S.GetStatus <> 0 then
    Halt;
  M.Process;
  M.Erase;
  M.Done;
end.
```

In this example, we load M from a stream created by the Store example using the stream's Get method, which calls Memo.Load indirectly. Note that the address of the buffer is registered with a code of 1000, as in the Store example.

You'll probably want to put in more error checking code than this example shows. See the BufIdStream example for proper error checking when reading/writing streams.

See Also

CommandWindow.Load Store

Declaration
```
function meOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if all of the specified options are currently selected.

Example
```
if M.meOptionsAreOn(meWordWrap) then
  M.meOptionsOff(meWordWrap)
else
  M.meOptionsOn(meWordWrap);
```
Toggle the word wrap option.

Declaration
```
procedure meOptionsOff(OptionFlags : LongInt);
```

Purpose
Turn options off.

Description
This method deactivates the specified option(s), excluding those designated as BadMemoOptions (see the Constants section, above).

Example
```
M.meOptionsOff(meWordWrap+meIndent);
```
Turns off word wrapping and auto-indenting.

Declaration
```
procedure meOptionsOn(OptionFlags : LongInt);
```

Purpose
Turn options on.

Description
This method activates the specified option(s), excluding those designated as BadMemoOptions (see the Constants section, above).

Example
```
M.meOptionsOn(meWordWrap+meIndent);
```
Turns on word wrapping and auto-indenting.

Process / ReinitBuffer

Declaration

```
procedure Process; virtual;
```

Purpose

Process editing commands.

Description

Memo's Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Editing Commands" at the start of this section.

Process calls the user-written status routine, if there is one, just prior to asking the user for the next command.

Example

See example at the beginning of this section.

See Also

CommandWindow.Process SetStatusProc

Declaration

```
procedure ReinitBuffer;
```

Purpose

Reinitialize internal variables when contents of buffer have changed.

Description

This method is used to inform the Memo editor that the contents of the text buffer have been changed. The most noticeable effect of calling it is that the cursor is moved to line 1, column 1.

ReinitBuffer is most typically needed in applications such as ENTRY.PAS, where the Memo editor is instantiated once and the contents of the text buffer can change from one Process call to the next. In such cases, the 'repeat Process until...' loop should be preceded by a call to ReinitBuffer.

Note that it is not necessary to call this method after calling MemoFile.ReadFile, because ReadFile makes the call itself after loading the new file.

Example

```
repeat
  M.Process;
  ExitCommand := M.GetLastCommand;
  case ExitCommand of
    ccQuit   : if M.meOptionsAreOn(meModified) then
                 {Save this memo};
    ccUser0  : begin
                 Buffer[1] := ^Z;
                 M.ReInitBuffer;
               end;
    {...other user exit commands...}
  end;
until (ExitCommand = ccQuit) or (ExitCommand = ccError);
```

In this somewhat atypical example, the text buffer is cleared and M's internal variables are reinitialized whenever the ccUser0 command is issued.

See Also

MemoFile.ReadFile

Declaration
```
procedure SetCtrlAttr(Color, Mono : Byte);
```

Purpose
Set attributes for control characters.

Description
This method allows you to set the attributes used to display control characters. If the meMapCtrls option is off, these attributes should be the same as those for ordinary text. If it is on, as it is by default, the attributes must be different, so that the user can distinguish ^A from 'A', etc.

Example
```
M.meOptionsOn(meMapCtrls);
M.SetCtrlAttr($03, $70);
```
Control characters will be displayed in cyan on black for color monitors, reverse video for monochrome monitors.

See Also
InitCustom

Declaration
```
procedure SetLineLimit(N : Integer);
```

Purpose
Set limit on number of lines.

Description
This method allows you to set a limit on the number of lines that may be entered by the user. The default limit is 32,767, which imposes no limit at all. (If the buffer can hold 65,521 bytes, one of which is reserved for a ^Z, the absolute maximum number of lines, if all of them were empty, would be 65,520/2 = 32,760.)

Note that the two reformatting commands (ccReformatP and ccReformatG) *do not* observe the limit imposed here. Why? Because forcing them to observe the limit would impose an unacceptable amount of overhead: the number of lines after reformatting would have to be calculated in advance, nearly doubling the time required to execute the commands. (Stopping the process at the point where the line limit was reached would be counterproductive, since reformatting the remaining lines might actually reduce the total.) This is usually not a problem, but it is certainly a potential problem in cases where the line limit must be strictly observed. In such cases, one solution would be to disable the reformatting commands by reassigning the appropriate key sequences to ccNone (see the documentation for OPCMD).

Similarly, when a MemoFile is being used to edit text loaded from a file, the line limit may be exceeded if one or more of the lines in the file have to be split because they exceed the maximum line length (see SetMaxLength).

Example
```
M.SetLineLimit(15);
```
Sets M's maximum number of lines to 15. If M has installed an ErrorProc (via SetErrorProc), that procedure will be called when you try to exceed 15 lines. If the default error procedure is in use it will just beep.

SetMaxLength / SetRightMargin

Declaration

```
procedure SetMaxLength(Len : Byte);
```

Purpose

Set maximum line length.

Description

This method allows you to specify the maximum length of a line of text in a text buffer. The default value, 254 (DefMaxLength), is the absolute maximum. If called at all, it should be called before the Memo window is first displayed. After the text in the buffer has been displayed once, it is unwise to change the maximum line length.

This setting has two effects. Most obviously, it determines the maximum number of characters that may be entered on a given line by the user, either by explicitly adding text to a given line or by combining lines. Less obviously, it can cause a line that is too long to begin with to be split as soon as the user moves the cursor onto that line. (This behavior is consistent with Borland's editors.) If possible, such lines will be split at the first word boundary at or before the limit; if not, the line will be split at the column just beyond the limit. In either case, a warning will be reported to the error routine (epWarning + ecLineTooLong). If the text buffer is already full at the time the line is split, the first two characters beyond the limit will be overwritten with a carriage return and line feed (^M^J), since the only viable alternative is to treat the error condition as fatal.

Important: When using this method, it is important to insure that the right margin is less than or equal to the maximum line length. See the entry for SetRightMargin.

Example

```
M.SetMaxLength(80);
```

Sets the maximum allowable line length to 80 characters.

Declaration

```
procedure SetRightMargin(RM : Byte);
```

Purpose

Set right margin.

Description

This method allows you to specify the column at which word wrap occurs. For example, if word wrap is on and the right margin is 78, then any attempt to enter new text (other than blank spaces) beyond column 78 will cause the line to be broken at the first word boundary at or before column 78. The default right margin is set by the Memo's constructor to be equal to the width of the edit window minus one. (E.g., if the edit window can display 80 columns, the default right margin would be 79.)

Very important: You should never allow the right margin to be set to 0 or to a value greater than the maximum length of a line (254 by default). Failure to observe this warning may lead to unpredictable results.

Example

```
M.SetRightMargin(70);
```

Sets the word wrap margin at column 70.

Declaration

```
procedure SetStatusProc(MSP : MemoStatusProc);
```

Purpose

Set status procedure.

Description

This method allows you to specify a user-written routine that is to be called just before MemoCommands.GetCommand is called to get the next editing command. The routine should be of the form

```
{$F+}
procedure MyStatusProc(MP : MemoPtr);
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure. The Memo/MemoFile/TextEditor making the call will always pass '@Self' as the MP parameter.

Be aware that OPMEMO supplies a "built-in" status routine, called MemoStatus, documented later in the chapter. You may use it as is, or you may use it as a model when writing your own.

Note that a TextEditor, an object ultimately derived from a Memo, will also call the status routine in two other situations: immediately after a new file has been loaded at the user's request (in case the status routine displays the name of the file being edited), and while searching for a string (to allow the user to follow the progress of the search). See EDITOR.PAS for an example of how to write a status routine for a TextEditor.

Examples

```
M.SetStatusProc(MemoStatus);
```

Activates OPMEMO's built-in status procedure, which displays one line of status information on line StatusRow (default = 2) in the default attribute (white on black for both color and mono displays).

```
StatusRow := ScreenHeight;
StatusColor := $0E;
StatusMono := $0F;
M.SetStatusProc(MemoStatus);
```

Uses the built-in status procedure but displays it at the bottom of the screen in either yellow on black (color displays) or white on black (mono displays).

See Also

MemoStatus

Declaration

```
procedure SetTabSize(TabSize : Byte);
```

Purpose

Set size for fixed tabs.

Description

This method allows you to specify the distance between tab stops. The default setting is 8 (DefTabDelta), which places tab stops at columns 1, 9, 17, etc.

Example

```
M.SetTabSize(5);
```

Sets tab stops at every fifth column (1, 6, 11, etc.).

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a Memo in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Memo to the stream S. Prior to calling this method, you should be certain that either 1) the address of the Memo's text buffer is registered, or 2) the meDeallocate option is selected. Besides telling the object's destructor to deallocate the text buffer, meDeallocate also tells Load to reallocate space for the buffer automatically when the object is loaded from the stream.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a Memo is MemoStream.

Example

```
uses  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const BufSize = 5000;
var
  M : Memo;
  S : BufIdStream;
  Status : Word;
  Buffer : array[1..BufSize] of Char;
begin
  Buffer[1] := ^Z;
  if not M.InitCustom(2, 4, 79, 24,                {Window coordinates}
                      DefaultColorSet,             {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      BufSize,                     {Buffer size}
                      @Buffer)                     {Buffer pointer}
  then begin
    WriteLn('Failed to init Memo. Status = ', InitStatus);
    Halt;
  end;
  S.Init('MEMO.STM', SCreate, 1024);
  S.RegisterHier(MemoStream);
  S.RegisterPointer(1000, @Buffer);
  S.Put(M);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  M.Done;
end.
```

Instantiates M, then stores it in a stream using the stream's Put method, which calls Memo.Store indirectly. Notice that the address of the buffer is registered with a code of 1000; any program that loads the Memo must therefore register a buffer of the same size using the same code.

See Also

CommandWindow.Store Load

2. The MemoFile

While reading the documentation for the Memo object, you may have wondered if it is possible to use a Memo for editing text files. Theoretically, the answer is Yes, but Memo does not provide any methods to assist in the reading and writing of text files. The MemoFile object does. In fact, the whole purpose of the MemoFile, a direct descendant of the Memo, is to provide a tiny layer of additional functionality that allows a Memo to be used for editing text files.

Compare the example program shown below to the one in the previous section.

```
program MemoFileExample; {EXMEMOF.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const BufSize = 65521;
var
  MF : MemoFile;
  FSize : LongInt;
  ExitCommand : Word;
begin
  {initialize MemoFile with custom window options}
  if not MF.InitCustomAndAlloc(2, 4, 79, 24,              {Window coordinates}
                               DefaultColorSet,           {ColorSet}
                               DefWindowOptions or wBordered, {Window options}
                               BufSize)                   {Buffer size}
  then begin
    WriteLn('Failed to init MemoFile. Status = ', InitStatus);
    Halt;
  end;

  {Use built-in status and error handlers}
  MF.SetStatusProc(MemoStatus);
  MF.SetErrorProc(MemoError);

  {create/read in a file}
  MF.ReadFile('TEMP.MEM', FSize);

  {assign ccSaveFile to <F2> and ccNewFile to <F3>}
  MemoCommands.AddCommand(ccSaveFile, 1, $3C00, 0);
  MemoCommands.AddCommand(ccNewFile,  1, $3D00, 0);
  repeat
    MF.Process;

    {Process exit commands}
    ExitCommand := MF.GetLastCommand;
    case ExitCommand of
      ccSaveFile : MF.SaveFile;
      ccQuit     : if MF.meOptionsAreOn(meModified) then
                     MF.SaveFile;
      ccNewFile  : begin
                     {...prompt for new filename, call ReadFile...}
                   end;
      {...user exit commands...}
    end;
  until (ExitCommand = ccQuit) or (ExitCommand = ccError);

  MF.Erase;
  MF.Done;
  ClrScr;
end.
```

If you took our advice and compared the two programs, you will have seen that there are really only two basic differences between them. First, the constructor used to instantiate the MemoFile takes one fewer parameter, because the constructor allocates the (much larger) text buffer itself. Although MemoFile provides constructors that require the extra parameter, it is generally more convenient to have the constructor allocate the buffer for you. Second, this program makes calls to two methods, ReadFile and SaveFile, that do not exist for a simple Memo. As their names imply, ReadFile reads the contents of an ASCII text file into the text buffer, and SaveFile saves the contents of the text buffer in that same file. The other differences between the two sample programs are all related in one way or another to the calls to these methods.

Three other points to be made concerning the example program shown above. First, its error handling is weak. Calls to ReadFile, in particular, should generally be followed by calls to GetLastError to determine whether or not the operation succeeded. Although it is perfectly acceptable to ask ReadFile to read a file that doesn't exist, it is not acceptable to continue if the name of the file is invalid. For an example of a program that uses MemoFile and contains better error checking, see MEMO.PAS. Second, we should point out that the reason for the calls to MemoCommands.AddCommand in the example program is that these two commands are not assigned to any default key sequences. Since these commands are typically needed in a program that uses a MemoFile, it is generally necessary to assign them to key sequences manually at runtime. And third, the example program does not actually implement the ccNewFile command, even though it assigns it a key sequence. To do so would have required adding quite a bit of code that would only be distracting. For an example of how the ccNewFile command should be processed, see MEMO.PAS.

Although the addition of the ReadFile and SaveFile methods represent the most obvious differences between a Memo and a MemoFile, there are others, principally in the category of options. Several of the options documented in the previous section pertain principally or exclusively to MemoFiles. meModified is a flag set when the text buffer is modified by the user; as shown in the example program above, this flag should usually be checked before destroying a MemoFile to determine whether or not SaveFile should be called. meAllowTrunc affects the behavior of ReadFile; it indicates that partial files may be read into the buffer in cases where the entire file won't fit. meMakeBackups affects the behavior of SaveFile; it indicates that, in cases where the file about to be written already exists, the existing version should first be renamed (with an extension of 'BAK'). meStreamReload affects the behavior of the Store and Load methods; it indicates that the name of the currently loaded file should be saved by Store, and that Load should attempt to reopen the file when it is called. meNewFile is a flag set by ReadFile to indicate to status routines that a new file has been loaded. And meNoCtrlZ also affects the behavior of SaveFile; it indicates that the ^Z character used to mark the end of the text buffer should not be written to disk.

Tab Characters

One important fact that we neglected to mention in our discussion of the Memo object is that it is not equipped to handle tab characters (^I) embedded in a text buffer. When you issue the ccTab command in a Memo editor while in insert mode, it does not insert a literal tab character; it inserts a series of blank spaces that effectively move the text that follows (if any) to the next tab stop. It's a minor point as far as the Memo is concerned, because typically any data being edited with the memo editor was created with it as well. It is an important issue for the MemoFile, however, because a MemoFile will often be used to edit text created with another editor, which may or may not use "real" tabs. The bottom line is that, if a file containing tab characters needs to be edited with a MemoFile--and this applies to its descendant, the

TextEditor, as well–that file should first be run through a conversion program that can expand the tabs to spaces. A very simple conversion program is shown in the documentation for the Detab function in OPSTRING.

Types

```
MemoFilePtr = ^MemoFile;
MemoFile =
  object(Memo)
    mfFileName : PathStr;   {name of file being edited}
  end;
```

A slightly modified Memo object to be used for editing text files.

Init / InitAndAlloc

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a memo window for editing files.

Description

This constructor instantiates a MemoFile using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
const
  BufSize = 50000;
var
  MF : MemoFile;
  Buffer : array[1..BufSize] of Char;
...
Buffer[1] := ^Z;
if not MF.Init(2, 4, 79, 24, BufSize, @Buffer) then begin
  WriteLn('Failed to init MemoFile. Status = ', InitStatus);
  Halt;
end;
```

Instantiates MF with default window colors and options.

See Also

InitCustom

Declaration

```
constructor InitAndAlloc(X1, Y1, X2, Y2 : Byte; BufferSize : Word);
```

Purpose

Create a memo window for editing files and allocate a buffer.

Description

This constructor instantiates a MemoFile using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). BufferSize indicates the desired size for the text buffer. See InitCustomAndAlloc and InitCustom for further details.

Example

```
const
  BufSize = 50000;
var
  MF : MemoFile;
...
if not MF.InitAndAlloc(2, 4, 79, 24, BufSize) then begin
  WriteLn('Failed to init MemoFile. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a MemoFile with a 50,000-byte buffer using default window colors and options.

See Also

InitCustomAndAlloc

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a memo window for editing files.

Description

This constructor instantiates a MemoFile using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window, where the text being edited will be displayed. BufferSize is the size of the text buffer to use, and BufPtr is its address. If the buffer is empty at the time InitCustom is called, there should be a ^Z in the first position.

InitCustom will fail only if the parent constructor (Memo.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Only two sets of attributes in the specified ColorSet have special relevance for window objects of type MemoFile. TextColor/TextMono determine the attribute used to display ordinary text, and CtrlColor/CtrlMono determine the attribute used to display control characters embedded in the text. After the object has been instantiated, these attributes may be changed by calling the SetTextAttr and SetCtrlAttr methods, respectively.

InitCustom initializes mfFileName to ''. It is generally assumed that ReadFile will be called between the time that InitCustom instantiates the object and Process is called to start processing commands. If ReadFile is not called, and mfFileName remains empty, calls to SaveFile will do nothing.

Example

```
const
  BufSize = 50000;
var
  MF : MemoFile;
  Buffer : array[1..BufSize] of Char;
...
Buffer[1] := ^Z;
if not InitCustom(2, 4, 79, 24,                {Window coordinates}
                  DefaultColorSet,             {ColorSet}
                  DefWindowOptions or wBordered, {Window options}
                  BufSize,                     {Buffer size}
                  @Buffer)                     {Buffer pointer}
then begin
  WriteLn('Failed to init MemoFile. Status = ', InitStatus);
  Halt;
end;
```

Instantiates MF with a bordered window and default colors.

See Also

InitCustomAndAlloc
Memo.InitCustom
ReadFile

InitCustomAndAlloc

Declaration

```
constructor InitCustomAndAlloc(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                               Options : LongInt; BufferSize : Word);
```

Purpose

Create a memo window for editing files and allocate a buffer.

Description

This constructor is identical to InitCustom, except that it allocates a text buffer of the specified size (BufferSize). If there is insufficient memory to allocate the buffer, InitCustomAndAlloc will fail, and InitStatus will be set to epFatal+ecOutOfMemory.

Example

```
const
  BufSize = 50000;
var
  MF : MemoFile;
...
if not MF.InitCustomAndAlloc(2, 4, 79, 24,                {Window coordinates}
                             DefaultColorSet,             {ColorSet}
                             DefWindowOptions or wBordered, {Window options}
                             BufSize)                     {Buffer size}
then begin
  WriteLn('Failed to init MemoFile. Status = ', InitStatus);
  Halt;
end;
```

Instantiates MF with a bordered window and default colors. We pass our desired size to MF and it allocates the heap space.

See Also

InitAndAlloc InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a MemoFile from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to MemoFile.Store. For a discussion of special issues involving the text buffer to be used for the MemoFile, see Memo.Load. (Typically a MemoFile will have a dynamically allocated buffer that needs to be reallocated by Load.)

If the meStreamReload option was in effect when the MemoFile was stored, the stream will contain the name of the file that was loaded at the time. If there is such a filename, Load will attempt to load that file. If it is unable to do so, Load will fail, and InitStatus will contain an I/O error code reported by ReadFile.

The stream registration procedure for a MemoFile is MemoFileStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const
  BufSize = 50000;
var
  MF : MemoFile;
  S : BufIdStream;
begin
  S.Init('MEMOFILE.STM', SOpen, 1024);
  S.RegisterHier(MemoFileStream);
  S.Get(MF);
  if S.GetStatus <> 0 then
    Halt;
  S.Done;
  MF.Process;
  MF.Erase;
  MF.Done;
end.
```

In this example, we load MF from a stream created by the Store example using the stream's Get method, which calls MemoFile.Load indirectly. Note that it is not necessary to call ReadFile here because the meStreamReload option was in effect when the object was stored.

See Also

Memo.Load Store

ReadFile / SaveFile

Declaration

```
procedure ReadFile(FName : string; var FSize : LongInt);
```

Purpose

Read a file into the text buffer.

Description

This routine reads a text file, FName, into the text buffer and stores the filename (in the mfFileName field) for future reference by SaveFile. If the file was opened successfully, FSize will have the size of the file on return, even if the file was not actually loaded. If the file doesn't exist and the path is valid, FSize will be set to 0, and the file will be created when SaveFile is called.

Any I/O errors that occur while reading an existing file are reported to the user-defined error routine, if there is one. Most I/O errors are classified as "fatal" by ReadFile.

If the file being read is larger than the text buffer, ReadFile will do one of two things, depending on the state of the meAllowTrunc option. If it is off, ReadFile will generate a fatal error (epFatal+ecFileTooLarge) and exit immediately. If it is on, it will read as much of the file as it can and generate a warning (epWarning+ecFileTruncated).

Example

```
MF.ReadFile('TEMP.MEM', FSize);
```

Reads in the file TEMP.MEM if it exists.

Declaration

```
procedure SaveFile;
```

Purpose

Save the text buffer in mfFileName.

Description

This routine saves the contents of the text buffer in the text file designated by mfFileName, a data field assumed to have been initialized by a previous call to ReadFile. If the file already exists and the meMakeBackups option is active, the existing file will be renamed to filename.BAK before the new file is created. If mfFileName is empty, SaveFile does nothing.

Any I/O errors that occur while saving the file are reported to the user-defined error routine, if there is one, and are classified as "non-fatal."

If the meNoCtrlZ option is off, as it is by default, the ^Z that marks the end of the text buffer will be written to disk. If it is on, the ^Z will not be written.

Example

```
MF.Process;
MF.meOptionsOn(meMakeBackups);
if MF.meOptionsAreOn(meModified) then
  MF.SaveFile;
```

Saves the current file only if that file was modified.

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a MemoFile in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the MemoFile to the stream S. If the meStreamReload option is in effect, the stream will also contain the name of the file that is currently being edited. In that case, when Load is called to reload the object, ReadFile will be called to try to reload the file into the editor. For a discussion of special issues involving the text buffer for the MemoFile, see Memo.Store.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a MemoFile is MemoFileStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo;
const
  BufSize = 50000;
var
  MF : MemoFile;
  S : BufIdStream;
  FSize : LongInt;
  Status : Word;
begin
  if not MF.InitCustomAndAlloc(2, 4, 79, 24, DefaultColorSet,
                               DefWindowOptions or wBordered, BufSize)
  then begin
    WriteLn('Failed to init MemoFile. Status = ', InitStatus);
    Halt;
  end;
  MF.ReadFile('TEMP.MEM', FSize);

  {Write MF out to a stream}
  MF.meOptionsOn(meStreamReload);
  S.Init('MEMOFILE.STM', SCreate, 1024);
  S.RegisterHier(MemoFileStream);
  S.Put(MF);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  MF.Done;
end.
```

Instantiates MF, loads a file, and stores MF in a stream using the stream's Put method, which calls Store indirectly. See Load for an example program that reloads this MemoFile.

See Also

Load Memo.Store

3. Procedures and Functions

This section documents ordinary procedures and functions that appear in OPMEMO, as well as related type, constant, and variable declarations.

Constants

```
ErrorRow    : Byte = 1;
ErrorColor  : Byte = $F;
ErrorMono   : Byte = $F;
```

These typed constants affect the behavior of MemoError. ErrorRow is the screen row where error messages are displayed. ErrorColor and ErrorMono determine the video attribute in which the messages are displayed on color and monochrome displays, respectively.

```
StatusRow   : Byte = 2;
StatusColor : Byte = $F;
StatusMono  : Byte = $F;
```

These typed constants affect the behavior of MemoStatus. StatusRow is the screen row where the status line is displayed. StatusColor and StatusMono determine the video attribute in which the status line is displayed on color and monochrome displays, respectively.

Declaration

```
procedure MemoError(UnitCode : Byte; var ErrorCode : Word; ErrorMsg : string);
```

Purpose

Display error message and wait for key press.

Description

This routine provides a ready-made error procedure for use with Memos and MemoFiles. Although it could also be used as an error handler for other object types, the assumptions it makes are best suited to a text editor of some sort. It does five things, essentially: 1) adds '. Press any key...' to the end of ErrorMsg; 2) displays the error message on ErrorRow (defaults to 1) in either ErrorColor or ErrorMono, depending on the current display; 3) moves the cursor to the end of the error message; 4) waits for the user to press a key; and 5) clears the line where the error message was displayed.

Examples

```
M.SetErrorProc(MemoError);
```

Display error messages at row ErrorRow (default = 1) in the default attribute (white on black for both color and monochrome displays).

```
M.SetErrorProc(MemoError);
ErrorRow := ScreenHeight;
ErrorColor := $0E;
ErrorMono := $0F;
```

Display error messages at the bottom of the screen in either yellow on black (color displays) or white on black (mono displays).

See Also

CommandWindow.SetErrorProc

Declaration

```
procedure MemoStatus(MP : MemoPtr);
```

Purpose

Display status line.

Description

This routine provides a ready-made status procedure for use with Memos and MemoFiles. The status line uses 80 columns, and is displayed at the row specified by the global typed constant StatusRow (defaults to 2) in either StatusColor or StatusMono, depending on the current display. The status line is of the form

```
Line: nnnnn  Column: nnn  100%  Insert: Off  Indent: Off  Word wrap: Off  SAVE
```

The first two fields show the position of the cursor. The third ('100%') shows the percentage of the text buffer that is currently in use (empty = 0%, full = 100%). The next three fields indicate the status of important mode toggles (insert mode, auto-indent mode, and word wrap). 'SAVE' appears when the text buffer has been marked as modified (i.e., the meModified flag is set).

Examples

```
StatusRow := ScreenHeight;
StatusColor := $0E;
StatusMono := $0F;
M.SetStatusProc(MemoStatus);
```

Display status line at the bottom of the screen in either yellow on black (color displays) or white on black (mono displays).

See Also

Memo.SetStatusProc

C. OPEDITOR: Advanced Text Editor

In describing the MemoFile object in the previous section, we indicated that its purpose was simply to extend the functionality of the Memo to permit the editing of text files. That's true, but what we didn't say was that the MemoFile was not the best tool in **Object Professional** for the job. The TextEditor object implemented in OPEDITOR is.

Derived from the MemoFile, the TextEditor provides a host of new editing commands, and the result is a plug-and-play editor comparable in power to the Turbo Pascal editor. Most of the new commands fall into one of four categories: file commands (for reading and writing text files), block commands (copy, move, delete, read, write, etc.), text marker commands (for setting place markers and returning to them), and search-and-replace commands (for locating text and making one-to-one string substitutions either locally or globally).

The simple example program shown below illustrates the basic steps involved in using a TextEditor.

```
program EditorExample; {EXEDITOR.PAS}
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor;
var
  TE : TextEditor;
  FSize : LongInt;
  ExitCommand : Word;
  AllDone : Boolean;
begin
  {instantiate a TextEditor with a bordered window}
  if not TE.InitCustom(2, 4, 79, 24,                {Window coordinates}
                       DefaultColorSet,             {ColorSet}
                       DefWindowOptions or wBordered, {Window options}
                       65521)                       {Buffer size}
  then begin
    WriteLn('Failed to init TextEditor. Status = ', InitStatus);
    Halt;
  end;

  {use built-in status and error handlers provided by OPMEMO}
  TE.SetStatusProc(MemoStatus);
  TE.SetErrorProc(MemoError);

  {Create/read in a text file}
  TE.ReadFile('READ.ME', FSize);

  AllDone := False;
  repeat
    TE.Process;
    ExitCommand := TE.GetLastCommand;
    case ExitCommand of
      ccSaveExit,           {Save and exit -- file already saved}
      ccAbandonFile,        {Abandon file}
      ccError :             {Fatal error}
        AllDone := True;
      {...user exit commands..}
    end;
  until AllDone;

  TE.Erase;
  TE.Done;
  ClrScr;
end.
```

With minor exceptions, this program is identical to the one shown in the previous section to illustrate the use of the MemoFile, and if you ran the two programs side-by-side you couldn't tell the difference. The differences are almost all internal. One difference you can see just by looking at the source code, however, is that the ccSaveFile command, which had to be processed manually when using a MemoFile, is handled automatically here. The ccNewFile command is also handled "automatically," but it requires the help of a user-written routine, as we'll explain later.

One other noteworthy difference between the two example programs is that the case statement that processes the exit command has to account for two new exit commands, ccSaveExit and ccAbandonFile, and that it does not check for the ccQuit command, which is not assigned to a key sequence in the default command table (EditorKeySet). The difference is a reflection of the fact that the TextEditor was designed specifically for the task of editing text files. The ccSaveExit command indicates that the user has asked to save the file (already done by Process) and then exit. The ccAbandonFile command indicates that the user has asked to quit without having his changes written to disk. (Most applications will view this request with skepticism and ask the user to verify the request if changes have been made.) Although it is comparable in meaning to the ccQuit command, it is assigned to a different key sequence than is normally used for ccQuit.

A noteworthy similarity is that this example program is equally negligent when it comes to error handling. For a better example, see EDITOR.PAS.

Editing Commands

OPEDITOR offers an extensive set of text editing commands, arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command.

Note that several of the commands in the list are enabled only if one or more user-written procedures are in place. In each of these cases, the entry for the command ends with the name(s) of the TextEditor method(s) that provide the relevant programming hook(s). The names appear within parentheses in italics, like so: *(SetEditProc)*.

Note too that all of the file-related commands are disabled if the TextEditor is being used as an advanced memo editor. (See the discussion of "TextEditors and Memo Fields," below.)

Cursor Movement

ccLeft **<Left>, <CtrlS>**
Cursor left one character. If the cursor is at the beginning of a line, and the teWrapAtLeft option is in effect, it is moved to the end of the previous line. If the option is not in effect, the cursor does not move.

ccRight **<Right>, <CtrlD>**
Cursor right one character.

ccWordLeft **<CtrlLeft>, <CtrlA>**
Cursor left one word. If the cursor is at the beginning of a line, it is moved to the end of the previous line.

ccWordRight **<CtrlRight>, <CtrlF>**
Cursor right one word. If the cursor is at the end of a line, it is moved to the beginning of the following line.

ccHome **<Home>, <CtrlQ><S>**
Cursor to beginning of line.

ccEnd **<End>, <CtrlQ><D>**
Cursor to end of line.

ccUp **<Up>, <CtrlE>**
Cursor up one line.

ccDown **<Down>, <CtrlX>**
Cursor down one line.

ccScrollUp **<CtrlW>**
Scroll display up one line.

ccScrollDown **CtrlZ>**
Scroll display down one line.

ccPageUp **<PgUp>, <CtrlR>**
Scroll display up one page.

ccPageDn **<PgDn>, <CtrlC>**
Scroll display down one page.

ccScreenTop **<CtrlHome>, <CtrlQ><E>**
Move cursor to top of edit window.

ccScreenBot **<CtrlEnd>, <CtrlQ><X>**
Move cursor to bottom of edit window.

ccTopOfFile **<CtrlPgUp>, <CtrlQ><R>**
Move cursor to beginning of file.

ccEndOfFile **<CtrlPgDn>, <CtrlQ><C>**
Move cursor to end of file.

ccJmpLine **<CtrlJ><L>**
Prompts the user for a line number, then moves the cursor to the specified line. If the line number specified is greater than the number of lines in the file, the cursor will be moved to the last line in the file. *(SetEditProc)*

Insertion and Deletion

ccDel **<Del>, <CtrlG>**
Delete character at cursor.

ccBack **<Bksp>, <CtrlH>, <CtrlBksp>**
Delete character to left of cursor. If the cursor is at the beginning of a line, the line will be joined with the previous line.

ccDelLine **<CtrlY>**
Delete current line.

ccDelEol **<CtrlQ><Y>**
Delete from cursor to end of line.

ccDelWord **<CtrlT>**
Delete word to right of cursor. If the cursor is at end of a line, the following line is joined with the current line.

ccSelect **<Enter>, <CtrlM>**
Start a new line.

ccInsertLine **<CtrlN>**
Insert a new line at the position of the cursor.

ccTab **<Tab>, <CtrlI>**
Move the cursor to the next tab stop. If insert mode is on, any text to the right of the cursor is moved to the right of the tab stop. When fixed tabs are in effect, the tab stops occur at 8-column intervals (by default). When smart tabs are in effect (the default setting), the tab stops are determined by the locations of the words on the previous line; the first character in each word represents a tab stop.

ccCtrlChar **<CtrlP>**
Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is not available if the video attribute for control characters is the same as that for ordinary text and the teMapCtrls option is on.

File Commands

ccSaveFile **<F2>, <CtrlK><S>**
Save the current file and continue editing.

ccNewFile **<F3>**
Load a new file. If the current file is modified, the user will first be asked if he wants to save it. *(SetGetFileProc, SetYesNoProc)*

ccAbandonFile **<CtrlK><Q>**
Quit editing; ignore any changes made to the text.

ccSaveSwitch **<CtrlK><D>**
Save the current file and load a new one. *(SetGetFileProc)*

ccSaveNamed **<CtrlK><N>**
Save the current file under a new name. *(SetGetFileProc)*

ccSaveExit **<CtrlK><X>**
Save the current file and exit.

Block Commands

ccBlkBegin **<F7>, <CtrlK><B>**
Mark the beginning of a block.

ccBlkEnd **<F8>, <CtrlK><K>**
Mark the end of a block.

ccBlkWord **<CtrlK><T>**
Mark the current word as a block.

ccJmpBegin **<CtrlQ><B>**
Jump to the beginning of the currently marked block.

ccJmpEnd **<CtrlQ><K>**
Jump to the end of the currently marked block.

ccBlkToggle **<CtrlK><H>**
Toggle the display of marked blocks. Note that several of the other block commands do nothing or generate errors if a marked block is not displayed.

ccBlkCopy **<CtrlK><C>**
Copy the currently marked and displayed block to the position of the cursor. The block markers are placed around the new copy of the block.

ccBlkMove **<CtrlK><V>**
Move the currently marked and displayed block to the position of the cursor. The block markers remain around the block at its new position.

ccBlkDelete **<CtrlK><Y>**
Delete the currently marked and displayed block.

ccBlkIndent **<CtrlK><I>**
Indent the currently marked and displayed block by a fixed number of spaces (defaults to 2). Does nothing if the cursor is not within the block.

ccBlkUnindent **<CtrlK><U>**
Unindent the currently marked and displayed block by a fixed number of spaces (defaults to 2). Does nothing if the cursor is not within the block.

ccBlkTCase **<CtrlO><O>**
Toggle the case of all characters in the currently marked and displayed block. If the cursor is not within a marked and displayed block, the case of the character at the cursor is toggled.

ccBlkUCase **<CtrlO><U>**
Convert all characters in the currently marked and displayed block to upper case. If the cursor is not within a marked and displayed block, the character at the cursor is converted to upper case.

ccBlkLCase **<CtrlO><V>**
Convert all characters in the currently marked and displayed block to lower case. If the cursor is not within a marked and displayed block, the character at the cursor is converted to lower case.

ccBlkRead **<CtrlK><R>**
Reads a file into the text buffer at the current position of the cursor and marks it as a block. If the entire contents of the file will not fit into the buffer, a warning will be generated (epWarning+ecBufferFull) and no data from the file will be read in. *(SetGetFileProc)*

ccBlkWrite **<CtrlK><W>**
Writes the currently marked and displayed block to a file. *(SetGetFileProc)*

ccBlkPrint **<CtrlK><P>**
Writes the currently marked and displayed block to the printer. By default, ccBlkPrint directs output to LPT1; to select LPT2 or LPT3, call the SetPrinter method.

Search and Replace

ccSearch **<CtrlQ><F>**
Allows the user to search for any string of up to 30 characters. After entering a search string, the user is prompted for search options, which are handled in the same fashion as the Borland editors. Valid search options are 'B' (search Backwards), 'G' (search Globally), 'L' (search Locally, within marked block), and 'U' (ignore case). If the search string is found, the text will be highlighted until the user presses a key. *(SetEditProc)*

ccReplace **<CtrlQ><A>**
This command works the same as the ccSearch command, except that the user is asked for a string of up to 30 characters to replace the search string with. In addition to the search options available for ccSearch, 'N' may be used to indicate that replacements are to be made without asking for confirmation. If 'N' is not selected, the user will be asked to confirm each replacement. When confirming a replacement, the user has four alternatives: yes (make the replacement), no (don't make it), all (make this replacement and continue making replacements without asking for confirmation), and quit (stop the whole operation). *(SetEditProc, SetYesNoProc)*

ccReSearch **<CtrlL>**
Repeat the last search or replace operation. The same string(s) used in the last search or replace operation will be used again, as will the previous search options. If no search operation has yet been performed, this command does nothing.

Text Markers

ccSetMark0..9 **<CtrlK><0>..<CtrlK><3>**
Sets one of the ten text markers at the position of the cursor. <CtrlK><0> sets marker 0, <CtrlK><1> sets marker 1, etc. Note that the default command table supplied by OPEDITOR does not assign any keys to ccSetMark4..ccSetMark9.

ccJmpMark0..9 **<CtrlQ><0>..<CtrlQ><3>**
Moves the cursor to a text marker set with one of the ccSetMarkX commands. <CtrlQ><0> jumps to marker 0, <CtrlQ><1> jumps to marker 1, etc. Note that the default command table supplied by OPEDITOR does not assign any keys to ccJmpMark4..ccJmpMark9.

ccMarkToggle **<CtrlK><M>**
Toggle the display of text markers.

ccPrevPos **<CtrlQ><P>**
Jump to the last cursor position before the cursor was moved to the current line. This command is especially useful following a search or replace operation.

Text Formatting

ccReformatP **<CtrlB>**
Reformat the current paragraph. Does nothing if word wrap is off.

ccReformatG **<CtrlK><G>**
Reformat the entire text buffer (Global reformat). Does nothing if word wrap is off. Use this command with caution!

ccCenterLine **<CtrlO><C>**
Center the current line between column 1 and the right margin.

Mode Toggles and Option Settings

ccIns **<Ins>, <CtrlV>**
Toggle insert mode on or off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

ccIndent **<CtrlO><I>**
Toggle auto-indent mode on or off. In auto-indent mode, pressing <Enter> while in insert mode will cause an inserted line to have the same indentation level as the previous line. Auto-indent mode also affects the way that text is formatted when word wrap occurs--the new line will have the same indentation level as the previous line--and hence the behavior of the reformatting commands (ccReformatP and ccReformatG).

ccWordWrap **<CtrlO><W>**
Toggle word wrap on or off. When word wrap is on, any attempt to insert or append text beyond the right margin will cause a new line to be inserted following the current line and all words that are at least partially beyond the right margin to be moved to the new line.

ccTabToggle **<CtrlO><F>**
Toggles the tab mode between smart tabs and fixed tabs.

ccTabSize **<CtrlO><T>**
Prompts the user for the number of columns between tab stops (1-10). *(SetEditProc)*

ccRtMargin **<CtrlO><R>**
Prompts the user for a new right margin. *(SetEditProc)*

ccSetIndent **<CtrlO><B>**
Prompts the user for the number of spaces to indent/unindent marked blocks when the ccBlkIndent and ccBlkUnindent commands are executed. The default value is 2. *(SetEditProc)*

Other

ccRestore **<CtrlQ><L>**
Restore original contents of the current line.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling EditorCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a help procedure, the TextEditor will pass it the following three parameters: ucEditor, @Self, and the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

ccMouseSel **<ClickLeft>**
Moves the cursor to the position indicated by the mouse. This command may also be used to scroll the edit window by clicking on a scroll bar.

Programming Hooks

The TextEditor provides two sets of hooks, beyond those available through methods inherited from Memo and CommandWindow, that allow it to be customized for a particular application.

The first is a trio of methods that take procedure pointers as parameters: SetEditProc, SetGetFileProc, and SetYesNoProc. Unlike most other hooks in **Object Professional**, these should not be considered optional. Any application that uses a TextEditor will want to take advantage of them. SetEditProc allows you to specify the procedure to be called when a string needs to be entered by the user. Without such a routine, several editing commands (notably

those involving search and replace operations) are effectively disabled. SetGetFileProc allows you to specify the routine to be called when a filename needs to be entered by the user. Without it, several of the file and block commands are disabled. And SetYesNoProc allows you to specify the routine to be called when an answer to a yes-no question needs to be obtained from the user. Without it, the ccNewFile command cannot ask the user to confirm the loss of any edits made to the current file, and the ccReplace command will behave oddly. For further details, see the individual entries for these three methods.

The second set of hooks consists of four methods: GotoLineCol, GotoOffset, InsertTextAtCursor, and InsertStringAtCursor. The purpose of these methods is to make it easier to implement user-written commands. GotoLineCol and GotoOffset allow you to force the cursor to be moved to a particular location in the file. GotoOffset might be used, for example, if you wanted to implement brace matching commands like the ones in the Turbo Pascal editor. You would scan the text buffer starting at the current location of the cursor (meBufPos+meCurCol-1) until you found the index for the character to move the cursor to, then pass that index to GotoOffset. GotoLineCol would be used in cases where the location needed to be expressed in terms of line numbers and screen columns. InsertTextAtCursor and InsertStringAtCursor provide a means of inserting text into the buffer at the current location of the cursor. You can use them to implement a simple macro system, for example, or a complex facility for retrieving "boilerplate" text from another file and inserting it into this one.

TextEditors and Memo Fields

Although the TextEditor was designed primarily to be used for editing text files, it can also be used for editing memo fields. There are three issues to keep in mind here, starting with the commands for reading and writing files (ccNewFile, ccSaveFile, etc.). These commands are all disabled when no file has been read into the text buffer with ReadFile or, more specifically, when the mfFileName field (inherited from MemoFile) is empty. The ccBlkRead and ccBlkWrite commands are the only file I/O commands that are not disabled; these commands can be used for importing and exporting data.

The second issue is that of the text buffer itself. The most commonly used constructors for a TextEditor, Init and InitCustom, both allocate a text buffer for you, making it difficult to edit text stored in a field in a record. So TextEditor provides two complementary constructors, InitBuf and InitBufCustom, that you allow you to specify your own text buffer.

Finally, there is the standard issue involving memo fields, that of dealing with changes to the contents of the text buffer. The solution here is the same as it is for a Memo: you must call ReinitBuffer when the contents of the text buffer change. Although TextEditor does not override the ReinitBuffer method itself, it does override a low-level method called by ReinitBuffer, giving it an opportunity to reinitialize various data fields specific to a TextEditor (those pertaining to block and text markers, for example).

Constants

```
BadEditorOptions : LongInt = teModified+teReformatting+teSearching+
                             teSearchStart+teSearchEnd+teHighlightOn+
                             teHighlightBack;
```

Options that exist for internal use, and may not be altered by calling teOptionsOn or teOptionsOff.

```
DefBlockIndent : Byte = 2;
```

Default indentation level for marked blocks.

```
DefEditorExt : ExtStr = '';
```

Default file extension to be used when prompting for filenames.

```
DefEditorOptions : LongInt = teInsert+teIndent+teWordWrap+teMapCtrls+
                             teMousePage+teMakeBackups+teSmartTabs+
                             teDeallocate;
```

The default options for a TextEditor.

```
EditorKeyMax = 400;
EditorKeyID : string[18] = 'opeditor key array';
EditorKeySet : array[0..EditorKeyMax] of Byte = (...);
EditorCfgEnd : Byte = 0;
```

EditorKeyId is an ID string used to mark the beginning of the configuration data area for OPEDITOR; EditorCfgEnd marks the end of the configuration data area. In between them is EditorKeySet, the command table used by EditorCommands. EditorKeyMax is the last valid index into the table.

```
MaxBlockIndent = 10;
```

Maximum indentation level for marked blocks.

```
MaxMarker = 9;
```

Maximum allowable position marker (first is 0).

```
MaxSearchLen = 30;
```

Maximum length of a search string.

```
MaxSearchOptions = 5;
teBackward  = 'B';
teNoCase    = 'U';
teGlobal    = 'G';
teNoConfirm = 'N';
teBlockOnly = 'L';
```

Search option characters. 'B' means search backward. 'U' means ignore case. 'G' means search globally, starting at the beginning (if searching forward) or end (if searching backward) of the file. 'N' means makes replacements without asking for confirmation. 'L' means search only within the currently marked and displayed block.

```
teInsert         = $00000001;  {True if in insert mode}
teIndent         = $00000002;  {True if in auto-indent mode}
teReadOnly       = $00000004;  {True if in read-only mode}
teWordWrap       = $00000008;  {True if word wrap is on}
teDeleteJoins    = $00000010;  {True if <Del> can join lines}
teModified       = $00000020;  {True if edits have been made}
teIndentIsPara   = $00000040;  {indent starts paragraph?}
teMousePage      = $00000080;  {clicking on scroll bar scrolls by page}
teAllowTrunc     = $00000100;  {read partial files?}
teMapCtrls       = $00000200;  {map control characters?}
teMakeBackups    = $00000400;  {make backup files?}
teStreamReload   = $00000800;  {reload previous file when activating a
                                text editor stored in a stream}
teReformatting   = $00001000;  {flag set while reformatting}
teNewFile        = $00002000;  {new file loaded}
teDeallocate     = $00004000;  {deallocate the buffer when Done is called?}
teWrapAtLeft     = $00008000;  {wrap to prev line on <Left> at column 1}
teNoCtrlZ        = $00010000;  {suppress ^Z when writing to disk}

teSmartTabs      = $01000000;  {smart tabs or fixed tabs?}
teSearching      = $02000000;  {flag set while searching}
teSearchStart    = $04000000;  {flag set at beginning of a search}
teSearchEnd      = $08000000;  {flag set at end of a search}
teHighlightOn    = $10000000;  {highlighting on}
teHighlightBack  = $20000000;  {highlighting goes backwards?}
teMarkersOn      = $40000000;  {text markers visible?}
teBlockOn        = $80000000;  {block markers on?}
```

These are the options that affect the behavior of a TextEditor. The first group of options, teInsert..teNoCtrlZ, are identical to the corresponding meXxx options for a Memo/MemoFile. See the documentation for OPMEMO for details. The second group, teSmartTabs..teBlockOn, are used only for TextEditors.

teSmartTabs indicates that smart tabs are to be used rather than fixed tabs. teSearching is a status flag that indicates that a search operation is currently in progress. teSearchStart is a flag set at the start of a search operation. teSearchEnd is a flag set at the end of a search operation. These flags make it possible for a user-written status routine to display a message indicating that a search is in progress, or to display a message at the end of a search-and-replace operation indicating the number of replacements that were made (the count is stored in the teReplacements field of the TextEditor). teHighlightOn and teHighlightBack are internal flags used when highlighting a string found in a search operation. teMarkersOn indicates that all text markers in use should be displayed. teBlockOn indicates that the currently marked block should be visible.

```
teNo        = 0;
teYes       = 1;
teQuit      = 2;
teAll       = 3;
```

Return codes for a user-written yes-no function (see EditorYesNoFunc).

Types

```
EditorEditFunc = function(MsgCode : Word; Prompt : string;
                          ForceUp, TrimBlanks : Boolean;
                          MaxLen : Byte; var S : string) : Boolean;
```

A user-written procedure to prompt the user for a string. (See entry for SetEditProc.)

```
EditorGetFileFunc = function(MsgCode : Word; Prompt : string;
                             ForceUp, TrimBlanks, Writing, MustExist : Boolean;
                             MaxLen : Byte; DefExt : ExtStr;
                             var S : string) : Boolean;
```

A user-written procedure to prompt the user for a filename. (See entry for SetGetFileProc.)

```
EditorYesNoFunc = function(MsgCode : Word; Prompt : string;
                           Default : Byte; QuitAndAll : Boolean) : Byte;
```

A user-written procedure to prompt the user for an answer to a yes-no question. (See the entry for SetYesNoProc, below.)

```
MarkerRec =
  record
    Line, Col : Integer;
  end;
MarkerArray = array[0..MaxMarker] of MarkerRec;
```

A MarkerRec is an internal data structure used to represent individual text and block markers. A MarkerArray is used to hold the entire group of text markers.

```
teSearchType = (tescNone, tescSearch, tescReplace);
```

A data type used internally to indicate the type of a search operation.

```
TextEditorPtr = ^TextEditor;
TextEditor =
  object(MemoFile)
    ...
    teSearchSt : string[MaxSearchLen];     {String to search for}
    teReplaceSt : string[MaxSearchLen];    {String to replace it with}
    teOptionSt : string[MaxSearchOptions]; {Search options}
    teLastSearch : teSearchType;           {Type of last search operation}
    teReplacements : Word;                 {Number of replacements made}
    teBlockIndent : Byte;                  {Indent level for blocks}
    teLPT : Byte;                          {line printer (1-3)}
    teDefExt : ExtStr;                     {Default extension}
    teBlkBegin : MarkerRec;                {Start of block}
    teBlkEnd : MarkerRec;                  {End of block}
    ...
  end;
```

A text editor. The data fields shown above should not be modified directly. They are documented here primarily to assist in the writing of status routines and menu-driven interfaces (see DESKPOP for examples).

teSearchSt is the last search string. teReplaceSt is the last replacement string. teOptionSt contains the search options last used. teLastSearch indicates the type of the last search operation. teReplacements indicates the number of replacements that were made in the last search-and-replace operation. A user-written status routine may want to report this information when it detects that the teSearchEnd flag is set. teBlockIndent is the indentation level used for the ccBlkIndent and ccBlkUnindent commands. teLPT is the line printer to use for the ccBlkPrint command (defaults to 1, meaning LPT1). teDefExt is the default file extension. teBlkBegin and teBlkEnd mark the start and end of the current block; the block is undefined if teBlkBegin.Line = 0 or teBlkEnd.Line = 0.

Variables

```
EditorCommands : CommandProcessor;
```

The default CommandProcessor for a TextEditor.

Declaration
```
procedure GotoLineCol(LineNum : Integer; Col : Byte);
```

Purpose
Move cursor to LineNum, Col.

Description
This method allows you to manually move the cursor to a particular location in the file. It might be used, for example, in a program that executes a compiler (such as Turbo Pascal) that reports the location of compile errors. The program could grab the error information off the screen, then load the file containing the error and use GotoLineCol to move the cursor to the appropriate location in the file.

The TextEditor must be the current window in order for this method to work. If the window is not current, it does nothing. If you do not want to display the edit window until the cursor has been positioned with GotoLineCol, you should draw the window initially on a virtual screen, like this:

```
TE.ActivateWrite;          {activate virtual screen}
TE.Draw;                   {draw window once}
TE.GotoLineCol(line, col); {position cursor}
TE.DeactivateWrite;        {deactivate virtual screen}
```

Example
```
TE.GotoLineCol(1, 1);
```
Moves the cursor to the top of the file.

Declaration
```
procedure GotoOffset(Offset : Word);
```

Purpose
Move cursor to specified location in the text buffer.

Description
This method is identical to GotoLineCol, except that the parameter is a particular Offset into the text buffer (a 1-based array of char), rather than a line and column. If Offset is invalid (less than 1 or greater than the number of characters currently in the buffer), it will be forced to a valid array index. For example, 'GotoOffset(65535)' would move the cursor to the end of the file, regardless of the buffer size; 'GotoOffset(0)' would move it to the top of the file.

Example
```
TE.GotoOffset(1);
```
Moves the cursor to the top of the file.

Init / InitBuf

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; BufferSize : Word);
```

Purpose

Create a window for editing files.

Description

This constructor instantiates a TextEditor using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if MaxAvail < 65521 then
  BufferSize := MaxAvail
else
  BufferSize := 65521;
if not TE.Init(1, 3, ScreenWidth, ScreenHeight, BufferSize) then begin
  WriteLn('Failed to init TextEditor. Status = ', InitStatus);
  Halt;
end;
```

Instantiates an unbordered text editor with the largest possible text buffer.

See Also

InitCustom

Declaration

```
constructor InitBuf(X1, Y1, X2, Y2 : Byte; BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a window for editing files, buffer already allocated.

Description

This constructor instantiates a TextEditor using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). BufferSize indicates the size for the text buffer, and BufPtr is its address. See InitBufCustom and InitCustom for further details.

Example

```
const
  BufSize = 20000;
var
  Buffer : array[1..BufSize] of Char;
  TE : TextEditor;
...
Buffer[1] := ^Z;
if not TE.InitBuf(1, 3, ScreenWidth, ScreenHeight, BufSize, @Buffer) then begin
  WriteLn('Failed to init TextEditor. Status = ', InitStatus);
  Halt;
end;
```

Instantiates an unbordered text editor with a pre-allocated buffer.

See Also

InitBufCustom

Declaration

```
constructor InitBufCustom(X1, Y1, X2, Y2 : Byte;
                          var Colors : ColorSet; Options : LongInt;
                          BufferSize : Word; BufPtr : Pointer);
```

Purpose

Create a custom window for editing files, buffer already allocated.

Description

This constructor is identical to InitCustom, except that it takes an additional parameter, BufPtr, which represents the address of the text buffer to use. If the buffer is empty at the time InitCustom is called, there should be a ^Z in the first position.

Example

```
const
  BufSize = 20000;
var
  Buffer : array[1..BufSize] of Char;
  TE : TextEditor;
...
Buffer[1] := ^Z;
if not TE.InitBufCustom(2, 4, 79, 24,                {Window coordinates}
                        DefaultColorSet,             {ColorSet}
                        DefWindowOptions or wBordered, {Window options}
                        BufferSize,                  {Buffer size}
                        @Buffer) then begin          {Buffer}
  WriteLn('Failed to init TextEditor. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a bordered text editor with a pre-allocated buffer.

See Also

InitCustom

InitCustom

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; BufferSize : Word);
```

Purpose

Create a custom window for editing files.

Description

This constructor instantiates a TextEditor using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window, where the text being edited will be displayed. BufferSize is the desired size of the text buffer, which will be allocated and initialized by the constructor.

InitCustom will fail only if the text buffer cannot be allocated or if the parent constructor (MemoFile.InitCustomAndAlloc) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Three sets of attributes in the specified ColorSet have special relevance for window objects of type TextEditor. BlockColor/BlockMono determine the attribute used to highlight marked blocks. MarkerColor/MarkerMono determine the attribute used to display text markers. And HighlightColor/HighlightMono determine the attribute used to temporarily highlight strings found by a search operation. After the object has been instantiated, these attributes may be changed by calling the SetBlockAttr, SetMarkerAttr and SetHighlightAttr methods, respectively.

Example

```
if MaxAvail < 65521 then
  BufferSize := MaxAvail
else
  BufferSize := 65521;
if not TE.InitCustom(2, 4, 79, 24,                {Window coordinates}
                     DefaultColorSet,             {ColorSet}
                     DefWindowOptions or wBordered, {Window options}
                     BufferSize)                  {Buffer size}
then begin
  WriteLn('Failed to init TextEditor. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a bordered, full-screen text editor with the largest possible text buffer.

See Also

Init
InitBufCustom
Memo.InitCustom
MemoFile.InitCustomAndAlloc

Declaration

```
procedure InsertStringAtCursor(S : String);
```

Purpose

Insert string S into the text buffer at the cursor.

Description

This method allows you to insert a string into the text buffer at the current position of the cursor. One possible use for it might be in a simple macro system, where pressing a certain key inserts a certain string into the text buffer. InsertStringAtCursor simply calls the more general InsertTextAtCursor method to perform the actual work of inserting the text.

Example

```
repeat
  TE.Process;
  ExitCommand := TE.GetLastCommand;
  case ExitCommand of
    ccAbandonFile,
    ccSaveExit,
    ccError : AllDone := True;
    ccUser0 : begin
                {...prompt for string to insert...}
                InsertStringAtCursor(S);
              end;
    {...user exit commands...}
  end;
until AllDone;
```

Shows how InsertStringAtCursor might be used as an exit command.

See Also

GotoLineCol

Declaration

```
procedure InsertTextAtCursor(var Data; Bytes : Word);
```

Purpose

Insert specified number of Bytes of Data into the text buffer at the cursor.

Description

This method allows you to insert Data into the text buffer at the current position of the cursor. One possible use for it might be in conjunction with a user exit command that displays a list of files containing "boilerplate" text that can be read in from disk and then merged into the file. After the text has been inserted into the buffer, the cursor will be moved to the first position beyond the text just inserted.

Data is treated as an array[1..Bytes] of Char, and it may contain anything an ordinary text file might contain, including carriage return/line feed combinations. Note that if Data contains a ^Z, only the characters that come before the ^Z will be merged into the text buffer.

The TextEditor must be the current window in order for this method to work. If the window is not current, it does nothing.

Example

See the example above.

Load / Process

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a TextEditor from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to TextEditor.Store. If the teStreamReload option was in effect when the TextEditor was stored, the stream will contain the name of the file that was loaded at the time. If there is such a filename, Load will attempt to load that file. If it is unable to do so, Load will fail, and InitStatus will contain an I/O error code reported by ReadFile.

The stream registration procedure for a TextEditor is TextEditorStream.

Example

```
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor;
var
  TE : TextEditor;
  S : BufIdStream;
begin
  S.Init('EDITOR.STM', SOpen, 1024);
  S.RegisterHier(TextEditorStream);
  S.Get(TE);
  if S.GetStatus <> 0 then Halt;
  S.Done;
  TE.Process;
  TE.Erase;
  TE.Done;
end.
```

In this example, we load TE from a stream created by the Store example using the stream's Get method, which calls TextEditor.Load indirectly. Note that it is not necessary to call ReadFile here because the teStreamReload option was in effect when the object was stored.

See Also

MemoFile.Load Store

Declaration

```
procedure Process; virtual;
```

Purpose

Process editing commands.

Description

The TextEditor's Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Editing Commands" at the start of this section.

Process calls the user-written status routine, if there is one, just prior to asking the user for the next command.

Example

See example at the beginning of this section.

See Also

CommandWindow.Process Memo.SetStatusProc

SetBlockAttr / SetBlockIndent / SetDefaultExtension

Declaration
```
procedure SetBlockAttr(Color, Mono : Byte);
```

Purpose
Set attributes for marked blocks.

Description
This method allows you to set the attribute used to highlight marked blocks.

Example
```
TE.SetBlockAttr($02, $70);
```
Displays marked blocks in green on black for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetBlockIndent(Indent : Byte);
```

Purpose
Set block indentation level.

Description
This method allows you to set the indentation level used by the ccBlkIndent and ccBlkUnindent commands. The default indentation level is 2.

Example
```
TE.SetBlockIndent(4);
```
Sets the indentation level to four columns.

Declaration
```
procedure SetDefaultExtension(DefExt : ExtStr);
```

Purpose
Default extension to use when prompting for filenames.

Description
This method sets the default extensions for filenames. ExtStr is a type ('string[3]') defined in OPSTRING.

Example
```
TE.SetDefaultExtension('DOC');
```
Sets the default extension for filenames to 'DOC'.

See Also
SetGetFileProc

SetEditProc

Declaration

```
procedure SetEditProc(EF : EditorEditFunc);
```

Purpose

Set edit function.

Description

This method allows you to designate a user-written routine that will be called when the TextEditor needs a string to be entered by the user. If such a routine is not present, certain editing commands will not be available (see the list of Editing Commands at the beginning of this section).

The routine should be of the form

```
{$F+}
function EditFunc(MsgCode : Word; Prompt : String;
                  ForceUp, TrimBlanks : Boolean;
                  MaxLen : Byte; var S : String) : Boolean;
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure.

MsgCode is the message code for the default Prompt. S is the string to be edited, and MaxLen is its maximum length. ForceUp indicates that the characters in the string should be forced to upper case. TrimBlanks indicates that leading and trailing spaces should be removed from the string. The function result should be True if the user entered anything, even if it is a blank string (as it might be when entering the replacement string for a search-and-replace operation).

Example

```
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor, OpEdit;
var TE : TextEditor;

  {$F+}
  function MyEditFunc(MsgCode : Word; Prompt : String;
                      ForceUp, TrimBlanks : Boolean;
                      MaxLen : Byte; var S : String) : Boolean;
  var LE : LineEditor;
  begin
    FastFill(80, ' ', 1, 1, TextAttr);
    LE.Init(DefaultColorSet);
    if ForceUp then
      LE.leEditOptionsOn(leForceUpper);
    if not TrimBlanks then
      LE.leEditOptionsOff(leTrimBlanks);
    LE.ReadString(Prompt, 1, 1, MaxLen, 40, S);
    MyEditFunc := (LE.GetLastCommand <> ccQuit);
    LE.Done;
    FastFill(80, ' ', 1, 1, TextAttr);
  end;
{$F-}
...
TE.SetEditProc(MyEditFunc);
```

Installs an edit function that displays prompts and accepts user input at the top of the screen. Note that the edit field width of 40 is somewhat arbitrary; in a real application the field width should be calculated based on the length of the prompt, MaxLen, and the space available.

See Also

SetGetFileProc SetYesNoProc

Declaration

```
procedure SetGetFileProc(GFF : EditorGetFileFunc);
```

Purpose

Set get file function.

Description

This method allows you to designate a user-written routine that will be called when the TextEditor needs a filename to be entered by the user. If such a routine is not present, certain editing commands will not be available (see the list of Editing Commands at the beginning of this section).

The routine should be of the form

```
{$F+}
function GetFile(MsgCode : Word; Prompt : string;
                 ForceUp, TrimBlanks, Writing, MustExist : Boolean;
                 MaxLen : Byte; DefExt : ExtStr;
                 var S : string) : Boolean;
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure.

MsgCode is the message code for the default Prompt. S is the string to be edited, and MaxLen is its maximum length. DefExt is the default extension to be applied to the filename, if any (ExtStr is declared in OPSTRING). ForceUp indicates that the characters in the string should be forced to upper case. TrimBlanks indicates that leading and trailing spaces should be removed from the string. If MustExist and Writing are both False, the user is being asked for the name of a new file that will be created later if necessary. If MustExist is True and Writing is False, the user is being asked for the name of an existing file that is to be merged into the current one. If Writing is True and MustExist is False, the user is being asked to enter the name of a file that will be overwritten if it already exists. The function result should be True only if the user entered the name of a file that meets the specified conditions.

Note that the default file extension, if there is one, will be applied to S by the calling routine if it hasn't been applied already by the user-written routine.

Example

```
uses
  OpCrt, OpDos, OpRoot, OpString, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor,
  OpEdit;
var
  TE : TextEditor;
  {$F+}
  function MyGetFileFunc(MsgCode : Word; Prompt : string;
                         ForceUp, TrimBlanks, Writing, MustExist : Boolean;
                         MaxLen : Byte; DefExt : ExtStr;
                         var S : string) : Boolean;
  var
    LE : LineEditor;
    Cmd : Word;
  begin
    MyGetFileFunc := False;
    FastFill(80, ' ', 1, 1, TextAttr);
    LE.Init(DefaultColorSet);
    if ForceUp then
      LE.leEditOptionsOn(leForceUpper);
    if not TrimBlanks then
      LE.leEditOptionsOff(leTrimBlanks);
    LE.ReadString(Prompt, 1, 1, MaxLen, 40, S);
```

```
    Cmd := LE.GetLastCommand;
    LE.Done;
    FastFill(80, ' ', 1, 1, TextAttr);
    if Cmd = ccQuit then
      Exit;
    S := DefaultExtension(S, DefExt);
    if Writing then
      if ExistFile(S) then
        {** confirm that user wants to overwrite file **}
      else
        MyGetFileFunc := True
    else if ExistFile(S) or not MustExist then
      MyGetFileFunc := True
    else
      {** error -- file doesn't exist **} ;
  end;
  {$F-}
...
TE.SetGetFileProc(MyGetFileFunc);
```

This is a pretty complete example of an EditorGetFileFunc. The chief things that are missing are 1) a calculated edit field width (40 is somewhat arbitrary), 2) a check to confirm that the user wants to overwrite a file (optional), and 3) an error message in the case where a file needs to exist and doesn't.

See Also

SetEditProc

Declaration
```
procedure SetHighlightAttr(Color, Mono : Byte);
```

Purpose
Set attributes for highlighted (found) text.

Description
This method lets you set the attribute used to display "highlighted" text. The attribute is used in two situations: 1) when a string is found by a search operation, and 2) when a search-and-replace operation is in progress and the user is being asked to confirm a particular replacement.

Example
```
TE.SetHighLightAttr($4F, $70);
```
Displays "found" text in white on red for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetMarkerAttr(Color, Mono : Byte);
```

Purpose
Set attributes for text markers.

Description
This method lets you specify the attribute used to display text markers. If the attribute is $70, for example, and the current display is a monochrome monitor, then text marker 1 would be represented as a '1' in reverse video.

Example
```
TE.SetMarker($1F, $70);
```
Displays place markers in white on blue for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetPrinter(LptNum : Byte);
```

Purpose
Set line printer (1-3).

Description
This method lets you designate the line printer to use for the ccBlkPrint command. LptNum should be a value in the range 1-3, corresponding to LPT1-LPT3. If LptNum is not in this range, LPT1 will be used.

Example
```
TE.SetPrinter(2);
```
Route printer output to LPT2.

SetYesNoProc

Declaration

```
procedure SetYesNoProc(YNF : EditorYesNoFunc);
```

Purpose

Set yes-no function.

Description

This method allows you to designate a user-written routine that will be called when the TextEditor needs an answer to a yes-no question. If such a routine is not present, certain editing commands will not be available (see the list of Editing Commands at the beginning of this section).

The routine should be of the form

```
{$F+}
function YesNo(MsgCode : Word; Prompt : string;
               Default : Byte; QuitAndAll : Boolean) : Byte;
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure.

MsgCode is the message code for the default Prompt. Default indicates the default response to the question is one of the following: teNo (0), teYes (1), teQuit (2), or teAll (3). These are also the only valid return codes for the function. Note that teAll is a valid response only if QuitAndAll is True. If QuitAndAll is False, teQuit is considered equivalent to teNo. QuitAndAll will be True only when a search-and-replace operation is in progress.

Example

```
uses
  OpCrt, OpString, OpRoot, OpCmd, OpWindow, OpMemo, OpEditor, OpField, OpEdit;
var TE : TextEditor;
  {$F+}
  function MyYesNo(MsgCode : Word; Prompt : string;
                   Default : Byte; QuitAndAll : Boolean) : Byte;
  var LE : LineEditor;
      Ch : Char;
      CharsToTake : CharSet;
  begin
    FastFill(80, ' ', 1, 1, TextAttr);
    LE.Init(DefaultColorSet);
    LE.leEditOptionsOn(leAllowEscape+leDefaultAccepted+leForceUpper);
    if Default = teYes then
      Ch := 'Y'
    else
      Ch := 'N';
    if QuitAndAll then begin
      CharsToTake := ['Y', 'N', 'A', 'Q'];
      Prompt := Prompt+' (Y/N/A/Q)'
    end
    else
      CharsToTake := ['Y', 'N'];
    LE.ReadChar(Prompt, 1, 1, CharsToTake, Ch);
    if LE.GetLastCommand = ccQuit then
      MyYesNo := teQuit
    else case Ch of
      'Y' : MyYesNo := teYes;
      'N' : MyYesNo := teNo;
      'A' : MyYesNo := teAll;
      'Q' : MyYesNo := teQuit;
    end;
```

```
    LE.Done;
    FastFill(80, ' ', 1, 1, TextAttr);
  end;
  {$F-}
...
TE.SetYesNoProc(MyYesNo);
```

A typical yes-no function. Notice that the LineEditor.ReadChar is used, rather than LineEditor.YesOrNo, so that Quit and All can be handled, as well as Yes and No.

See Also

SetEditProc

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a TextEditor in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the TextEditor to the stream S. If the teStreamReload option is in effect, the stream will also contain the name of the file that is currently being edited. In that case, when Load is called to reload the object, ReadFile will be called to try to reload the file into the editor.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a TextEditor is TextEditorStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpMemo, OpEditor;
var
  TE : TextEditor;
  S : BufIdStream;
  FSize : LongInt;
  Status : Word;
begin
  if not TE.InitCustom(2, 3, 79, 24, DefaultColorSet,
                       DefWindowOptions or wBordered, 65521) then begin
    WriteLn('Failed to init TextEditor. Status = ', InitStatus);
    Halt;
  end;
  TE.ReadFile('READ.ME', FSize);
  TE.teOptionsOn(teStreamReload);
  S.Init('EDITOR.STM', SCreate, 1024);
  S.RegisterHier(TextEditorStream);
  S.Put(TE);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  TE.Done;
end.
```

Instantiates TE, loads a file, and stores TE in a stream using the stream's Put method, which calls Store indirectly. See Load for an example program that reloads this TextEditor.

See Also

Load MemoFile.Store

Declaration
```
function teOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if the specified option(s) are currently selected.

Example
```
if TE.teOptionsAreOn(teWordWrap) then
  TE.teOptionsOff(teWordWrap)
else
  TE.teOptionsOn(teWordWrap);
```
Toggle the word wrap option.

Declaration
```
procedure teOptionsOff(OptionFlags : LongInt);
```

Purpose
Turn specified option(s) off.

Description
This method deactivates the specified option(s), excluding those designated as BadEditorOptions (see the Constants section, above).

Example
```
TE.teOptionsOff(teWordWrap+teIndent);
```
Turns off word wrapping and auto-indenting.

Declaration
```
procedure teOptionsOn(OptionFlags : LongInt);
```

Purpose
Turn specified option(s) on.

Description
This method activates the specified option(s), excluding those designated as BadEditorOptions (see the Constants section, above).

Example
```
TE.teOptionsOn(teWordWrap+teIndent);
```
Turns on word wrapping and auto-indenting.

D. OPBROWSE: File Browser

This section is divided into two parts. The first covers the Browser object, and the second covers the ordinary procedures and functions that are interfaced by OPBROWSE.

1. The Browser

When reading the documentation for the TextEditor object, it may have occurred to you that by selecting the teReadOnly option you could use the TextEditor as a "file browser," a utility for examining the contents of a file without changing it. Although that trick would certainly work, the result would be a rather limited file browser: it wouldn't work with binary files, nor would it work with any file larger than 64K. These limitations are generally acceptable for an editor, but far less so for a file browser. Hence the need for the Browser object.

The Browser takes an entirely different approach to storing the contents of a file. Rather than storing the data in a single, usually large text buffer, the Browser stores data in multiple 4K buffers, called "page buffers." More importantly, it doesn't insist on storing the entire contents of the file in memory at one time. Instead, it maintains a group of data structures known as "pages," one for each page buffer, that keep track of which sections of the file are currently loaded in the page buffers. If the Browser needs access to a part of the file that is not in memory, it will examine the list of pages, find one that seems to be in little demand, then overwrite its buffer with the data that is needed at the moment. As a result, a Browser can (at least theoretically) be used to examine a file as large as 2 gigabytes while using as little as 8K of buffer space. (Obviously, the greater the buffer space, the better the performance.)

The little example program shown below illustrates the basic steps involved in using a Browser.

```
program BrowseExample; {EXBROWSE.PAS}
uses
  Dos, OpCrt, OpRoot, OpCmd, OpDos, OpFrame, OpWindow, OpBrowse;
var
  B : Browser;
  Status : Word;
  Finished : Boolean;
begin
  if not B.InitCustom(2, 3, ScreenWidth-1, ScreenHeight-1,   {Coordinates}
                      DefaultColorSet,                       {ColorSet}
                      DefWindowOptions or wBordered,         {Window options}
                      MaxAvail)                              {Max heap space to use}
  then begin
    WriteLn('Failed to init Browser. Status = ', InitStatus);
    Halt;
  end;

  {Open a file, check for success}
  B.OpenFile('BROWSE.PAS');
  Status := B.GetLastError;
  if Status <> 0 then begin
    WriteLn('Failed to load file. Status = ', Status);
    Halt;
  end;

  {use built-in status routine}
  B.SetStatusProc(BrowseStatus);

  Finished := False;
```

```
  repeat
    B.Process;
    case B.GetLastCommand of
      ccQuit, ccError: Finished := True;
      {...user exit commands...}
    end;
  until Finished;

  B.Erase;
  B.Done;
  ClrScr;
end.
```

As you can see, the first six parameters to InitCustom are standard fare: window coordinates, colors, and window options. The last parameter is unique to Browsers. It specifies the maximum amount of memory that the Browser is allowed to use for pages and page buffers. Because our program is a simple one, and no other heap space needs to be allocated once the Browser has been instantiated, we give it the largest possible amount, MaxAvail.

Once the Browser has been instantiated, our next step is to give it a file to browse, so we call the OpenFile method, which opens the file for reading (only). In contrast to the example programs we showed you for the MemoFile and TextEditor objects, this one is a little better about error checking–because it has to be. If an error occurs while trying to open the file, there's no point in continuing, because Process cannot do anything except abort with an exit command of ccError.

Before entering the main program loop, we also add a call to SetStatusProc to establish a status routine, a procedure that will display a status line at the top of the screen (notice that the call to InitCustom would place the top of the frame on row 2, to leave room for the status line). The parameter is BrowseStatus, a status routine supplied by OPBROWSE that we can use unmodified, or use as a model for writing a custom status routine.

The main program loop is a simple one, because there are no exit commands to be handled other than the usual ones, ccQuit (quit browsing) and ccError (fatal error occurred).

Like most of our sample programs, this one ends with basic cleanup code. The window is erased, the destructor is called to close the file being browsed, and the entire screen is cleared to get rid of any text on the top row of the screen.

There are many ways to improve upon this program. It has no error reporting routine, for example. More importantly, it does not give the user access to the full range of browser commands. To do so, it would have to provide routines that allow strings and filenames to be edited. This issue is discussed in the next two sections, on "Browser Commands" and "Programming Hooks." If you want to see a more comprehensive example program, take a look at BROWSE.PAS.

Browser Commands

The commands available while browsing through a file are arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command.

Note that several of the commands in the list are enabled only if one or more user-written procedures are in place. In each of these cases, the entry for the command ends with the

name(s) of the Browser method(s) that provide the relevant programming hook(s). The names appear within parentheses in italics, like so: *(SetEditProc)*.

Cursor Movement

ccLeft **<Left>, <CtrlS>**
Scroll window left 1 column.

ccRight **<Right>, <CtrlD>**
Scroll window right 1 column.

ccWordLeft **<CtrlLeft>, <CtrlA>**
Scroll window left 10 columns.

ccWordRight **<CtrlRight>, <CtrlF>**
Scroll window right 10 columns.

ccHome **<Home>, <CtrlQ><S>**
Scroll window to column 1.

ccEnd **<End>, <CtrlQ><D>**
Scroll window to leftmost column, so that the end of the longest line on the screen is displayed.

ccUp **<Up>, <CtrlE>, <CtrlW>**
Scroll window up one line.

ccDown **<Down>, <CtrlX>, <CtrlZ>**
Scroll window down one line.

ccPageUp **<PgUp>, <CtrlR>**
Scroll window up one page.

ccPageDn **<PgDn>, <CtrlC>**
Scroll window down one page.

ccTopOfFile **<CtrlPgUp>, <CtrlQ><R>**
Scroll to beginning of file.

ccEndOfFile **<CtrlPgDn>, <CtrlQ><C>**
Scroll to end of file.

ccJmpLine **<CtrlJ><L>**
Prompts the user for a line number, then scrolls the window such that the specified line is at the top of the window. If the line number specified is greater than the number of lines in the file, the window will be scrolled to the end of the file. *(SetEditProc)*

File Commands

ccNewFile **<F3>, <CtrlK><N>**
Load a new file into the browser. *(SetGetFileProc)*

Block Commands

ccBlkBegin **<F7>, <CtrlK><B>**
Marks the line at the top of the window as the start of a block. Note that the browser allows only whole lines to be part of a block.

ccBlkEnd **<F8>, <CtrlK><K>**
Marks the line at the top of the window as the end of a block.

ccBlkBottom **<CtrlB><K>**
Marks the line at the bottom of the window as the end of a block.

ccJmpBegin **<CtrlQ><B>**
Jump to the beginning of the currently marked block.

ccJmpEnd **<CtrlQ><K>**
Jump to the end of the currently marked block.

ccBlkToggle **<CtrlK><H>**
Toggle the display of marked blocks. Note that the ccBlkWrite and ccBlkPrint commands both generate errors if a marked block is not displayed.

ccBlkWrite **<CtrlK><W>**
Writes the currently marked and displayed block to a file. Note that this command writes the actual contents of the file being browsed, not the image of it that is being displayed on the screen. The command does the same thing whether in ASCII or hex mode. *(SetGetFileProc)*

ccBlkPrint **<CtrlK><P>**
Writes the currently marked and displayed block to the printer. As with the ccBlkWrite command, this option writes the actual contents of the file being browsed to the printer. By default, ccBlkPrint directs output to LPT1; to select LPT2 or LPT3, call the SetPrinter method.

Search

ccSearch **<CtrlQ><F>**
Allows the user to search for any string of up to 30 characters. After entering a search string, the user is prompted for search options, which are handled in the same fashion as the Borland editors. Valid search options are 'B' (search Backwards), 'G' (search Globally), and 'U' (ignore case). If the search string is found, the line where the text is found will be highlighted until another search operation is performed or the mode is toggled between hex and ASCII. *(SetEditProc)*

ccReSearch **<CtrlL>**
Repeat the last search operation. The string used in the last search operation will be used again, as will the previous search options. If no search operation has yet been performed, this command does nothing.

Text Markers

ccSetMark0..9 **<CtrlK><0>..<CtrlK><3>**
Sets one of the ten text markers at the start of the line at the top of the window. <CtrlK><0> sets marker 0, <CtrlK><1> sets marker 1, etc. Note that the default command table supplied by OPBROWSE does not assign any keys to ccSetMark4..ccSetMark9.

ccJmpMark0..9 **<CtrlQ><0>..<CtrlQ><3>**
Scrolls the window such that the line associated with the specified text marker is at the top of the window. <CtrlQ><0> jumps to marker 0, <CtrlQ><1> jumps to marker 1, etc. Note that the default command table supplied by OPBROWSE does not assign any keys to ccJmpMark4..ccJmpMark9.

Mode Toggles

ccHexMode **<CtrlH>**
Toggles between ASCII mode (the default) and hex mode. In hex mode, either 8 or 16 characters are displayed per line, depending on the width of the window. (If the inner portion of the window is < = 50 columns 8 characters are displayed per line). The leftmost column displays the offset within the file (in hex) for the first character in the "line," the middle columns display the contents of the line in hex, and the rightmost column displays the contents of the line in ASCII.

ccStripHigh **<CtrlQ><H>**
Activates or deactivates the option to suppress (strip) the eighth (high) bit for each character in the file. In ASCII mode, this option affects only characters with an ordinal value of 128 or higher (10000000 in binary), causing #228, for example, to be displayed as #100 ('d'). In hex mode, it affects control characters as well, causing them to be represented as periods ('.'). The hex values in the middle columns are not affected by this option.

ccTabExpand **<CtrlQ><T>**
Enables or disables tab expansion. When this option is on, tab characters (^I) are expanded to spaces, aligning the text that follows at the next tab stop. (Tab stops occur every 8 columns: 1, 9, 17, etc.) This option does nothing in hex mode.

Other

ccQuit **<CtrlBreak>, <Esc>, <ClickRight>**
Quit browsing.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling BrowseCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a help procedure, the Browser will pass it the following three parameters: ucBrowse, @Self, and the value designated as the window's help index (see CommandWindow.SetHelpIndex), which defaults to 0.

ccMouseSel **<ClickLeft>**
This command does nothing unless the browse window has a scroll bar or mouse hot spots associated with it.

Programming Hooks

The programming hooks provided by Browser are similar to those available for a TextEditor, and again they fall into two categories.

The first category consists of methods that take procedure pointers as parameters: SetStatusProc, SetEditProc, and SetGetFileProc. SetStatusProc allows you to specify a routine that will be called when the display is scrolled or a potentially time-consuming operation begins. As shown in the example program above, OPBROWSE provides a built-in status routine that will suffice in many instances. SetEditProc allows you to specify the procedure to be called when a string needs to be entered by the user. Without such a routine, several commands (notably those involving search and replace operations) are effectively disabled. And SetGetFileProc allows you to specify the routine to be called when a filename needs to be entered by the user. Without it, several of the file and block commands are disabled. Like their counterparts in TextEditor, these last two hooks should be used in virtually any program that uses a Browser.

The second category consists of three methods that make it easier to implement user-written commands: GotoLine, GotoOffset, and Transfer. GotoLine and GotoOffset allow you to force the display to be scrolled to a particular location in the file being browsed. GotoOffset is generally more useful when browsing binary files in hex mode, while GotoLine is more useful when browsing text files in ASCII mode. Transfer provides a means of extracting data from the file being browsed. A typical application for it would be in a program that can display both a file browser and a text editor (such as DESKPOP). Transfer could be used to implement a user-exit command that copies the contents of the currently marked block from the Browser into the TextEditor.

Constants

```
BadBrowseOptions : Word = 0;
```

Options that exist for internal use, and may not be altered by calling brOptionsOn or brOptionsOff. No options currently fall into this category.

```
brBackward  = 'B';
brNoCase    = 'U';
brGlobal    = 'G';
```

Search option characters. 'B' means search backward. 'U' means ignore case. 'G' means search globally, starting at the beginning (if searching forward) or end (if searching backward) of the file.

```
brHexMode       = $0001;  {hex mode}
brTabExpand     = $0002;  {tab expansion}
brStripHigh     = $0004;  {strip high bits (except hex mode)}
brMousePage     = $0008;  {clicking on scroll bar scrolls by page}
brNewFile       = $0010;  {flag set when a new file is loaded}
brStreamReload  = $0020;  {reload previous file when activating a browser
                           stored in a stream}
```

These are the options that affect the behavior of a Browser. brHexMode selects hex mode, in which the file is displayed in a hex dump format, rather than ASCII mode. brTabExpand enables the expansion of tab characters to spaces in ASCII mode. brStripHigh causes the high bit of each character to be stripped (for display purposes only). brMousePage determines what happens when the user clicks on the up and down arrows of a scroll bar. If the option is off, the browse window will be scrolled up or down by a single line; if it is on, the browse window will be scrolled up or down one full page. brNewFile is an internal flag that is set after a new file has been loaded; its sole purpose is to allow status routines to determine when the current filename has changed. brStreamReload affects the behavior of Browser.Load and Browser.Store. If it is on, Store saves the name of the file currently being browsed in the stream. The next time that Load is called, it will attempt to reload the file whose name was saved.

```
BrowseKeyMax = 260;
BrowseKeyID  : string[13] = 'opbrowse keys';
BrowseKeySet : array[0..BrowseKeyMax] of Byte = (...);
BrowseCfgEnd : Byte = 0;
```

BrowseKeyId is an ID string used to mark the beginning of the configuration data area for OPBROWSE; BrowseCfgEnd marks the end of the configuration data area. In between them is BrowseKeySet, the command table used by BrowseCommands. BrowseKeyMax is the last valid index into the table.

```
DefBrowseExt : ExtStr = '';
```

Default file extension to be used when prompting for filenames.

```
DefBrowseOptions : Word = brTabExpand+brMousePage;
```

The default options for a Browser.

```
MaxScrollCol : Word = 999;
```

Maximum column for horizontal scrolling. (999 is arbitrary and may be adjusted). The value is 0-based, meaning that if MaxScrollCol is 999 then the last column that may be displayed at the left edge of the screen would be 1000. MaxScrollCol = 0 would disable horizontal scrolling.

```
MaxSearchLen = 30;
```

Maximum length of a search string.

Types

```
BrowseEditFunc = function (MsgCode : Word; Prompt : string;
                           ForceUp, TrimBlanks : Boolean;
                           MaxLen : Byte; var S : string) : Boolean;
```

A user-written procedure that prompts the user for a string. (See the entry for SetEditProc, below.)

```
BrowseGetFileFunc = function (MsgCode : Word; Prompt : string;
                              ForceUp, TrimBlanks : Boolean;
                              Writing, MustExist : Boolean;
                              MaxLen : Byte; DefExt : ExtStr;
                              var S : string) : Boolean;
```

A user-written procedure that prompts the user for a filename. (See the entry for SetGetFileProc, below.)

```
BrowserPtr = ^Browser;
Browser =
  object(CommandWindow)
    ...
    brLPT : Byte;               {line printer (1-3)}
    brDefExt : ExtStr;          {default extension}
    brSearchSt : string[MaxSearchLen]; {String to search for}
    brOptionSt : string[5];     {Search options}
    brLastPos : LongInt;        {Highest file offset = size-1}
    brCurPos : LongInt;         {Current display position in file}
    brCurLine : Word;           {Current display line in file, 1..Max}
    brLastLine : Word;          {Last line number in file}
    brLastLine2 : Word;         {Last line number in secondary mode (hex)}
    brColOfs : Integer;         {Horizontal scroll offset, 0..MaxCol}
    brBlkBegin : MarkerRec;     {Start of block}
    brBlkEnd : MarkerRec;       {End of block}
    brWorkingFlag : ShortInt;   {non-0 when time-consuming routine in progress}
    brBlockOn : Boolean;        {True when block is visible}
    ...
  end;
```

A file browser. The data fields shown above should not be modified directly. They are documented here primarily to assist in the writing of status routines and menu-driven interfaces (see DESKPOP for examples).

brLPT is the line printer to use for the ccBlkPrint command (defaults to 1, meaning LPT1). brDefExt is the default file extension. brSearchSt is the last search string. brOptionSt contains the search options last used. brLastPos is the last valid offset in the file (FileSize-1). brCurPos is the file offset for the line currently displayed at the top of the browse window. brCurLine is its line number (means different things in hex and ASCII modes). brLastLine is the line number for the last line in the file (0 if unknown). If a mode change (ASCII to hex) has occurred, brLastLine2 has the value of brLastLine before the mode change. brColOfs is the column currently displayed at the left edge of the window minus 1.

brBlkBegin and brBlkEnd contain the locations for the start and end of a marked block. The block is undefined if brBlkBegin.FilePos < = 0 or brBlkBegin.FilePos > = brBlkEnd.FilePos. brWorkingFlag is set to a non-0 value when a time consuming operation is in progress (such as a search) or new data has to be read in from disk. brBlockOn is an internal flag set to True when marked blocks are visible.

```
BrowseStatusProc = procedure(BP : BrowserPtr);
```

A user-written routine that displays status information (current line, column, etc.). BP is a pointer to the Browser that is making the call.

```
MarkerRec =
  record
    FilePos : LongInt;
    LineNum : Word;
  end;
MarkerArray = array[0..MaxMarker] of MarkerRec;
```

A MarkerRec is an internal data structure used to represent individual text and block markers. A MarkerArray is used to hold the entire group of text markers.

Variables

```
BrowseCommands : CommandProcessor;
```

The default CommandProcessor for a Browser.

brOptionsAreOn / brOptionsOff / brOptionsOn

Declaration
```
function brOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if the specified option(s) are currently selected.

Example
```
if B.brOptionsAreOn(brStripHigh) then
  B.ToggleHighBitStripping;
```
Forces high-bit stripping off.

Declaration
```
procedure brOptionsOff(OptionFlags : Word);
```

Purpose
Turn specified option(s) off.

Description
This method deactivates the specified option(s), excluding those designated as BadBrowseOptions (see the Constants section, above).

Important: Although the brStripHigh and brHexMode options are not designated as "bad," you should never use brOptionsOff to change them while a file is loaded into the Browser. Use ToggleHighBitStripping and/or ToggleModes instead.

Example
```
B.brOptionsOff(brTabExpand);
```
Turns off option to expand tabs to spaces.

Declaration
```
procedure brOptionsOn(OptionFlags : Word);
```

Purpose
Turn specified option(s) on.

Description
This method activates the specified option(s), excluding those designated as BadBrowseOptions (see the Constants section, above).

Important: Although the brStripHigh and brHexMode options are not designated as "bad," you should never use brOptionsOn to change them while a file is loaded into the Browser. Use ToggleHighBitStripping and/or ToggleModes instead.

Example
```
B.brOptionsOn(brTabExpand);
```
Turns on option to expand tabs to spaces.

See Also
ToggleHighBitStripping ToggleModes

Declaration
```
procedure CloseFile;
```

Purpose
Close the current browse file.

Description
This method closes the file currently loaded into the browser. It rarely needs to be called, since Done calls it automatically when an object is destroyed, and OpenFile calls it before opening a new file. One example of a case where it might be needed is a program that needs to temporarily close the file, Exec another program that uses that file, and then reopen the same file.

Example
See the example for GotoLine.

See Also
OpenFile

Declaration
```
function CurrentFileName : String;
```

Purpose
Return the name of the current browse file.

Description
This method returns the name of the file currently loaded into the browser, if any. It is typically used only by status routines.

Example
```
WriteLn('Browsing file: ', B.CurrentFileName);
```
Display the name of the current file.

See Also
SetStatusProc

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate page buffers.

Description
This destructor disposes of all memory allocated for page buffers and then calls the parent's destructor to deallocate memory used for window buffers, shadows, etc.

See Also
CommandWindow.Done
Init
InitCustom

GotoLine / GotoOffset

Declaration

```
procedure GotoLine(Line : Word);
```

Purpose

Scroll display to Line.

Description

This method lets you scroll the display such that a particular Line appears at the top of the browse window.

Example

```
SaveLine := B.brCurLine;
SaveName := B.CurrentFileName;
B.Erase;
B.CloseFile;
{...do something...}
B.OpenFile(SaveName);
B.GotoLine(SaveLine);
B.Draw;
```

Close the current file, do something, reopen the file and jump to the same line that was at the top of the browse window before. (Error checking is omitted here.)

Declaration

```
procedure GotoOffset(Offset : LongInt);
```

Purpose

Scroll display to the line nearest Offset.

Description

This method lets you scroll the display such that the line nearest a particular file Offset appears at the top of the browse window.

Example

```
SavePos := B.brCurPos;
SaveName := B.CurrentFileName;
B.Erase;
B.CloseFile;
{...do something...}
B.OpenFile(SaveName);
B.GotoOffset(SavePos);
B.Draw;
```

Close the current file, do something, reopen the file and jump to the same line that was at the top of the browse window before. (Error checking is omitted here.)

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte; HeapToUse : LongInt);
```

Purpose

Initialize a Browser with default options and colors.

Description

This constructor instantiates a Browser using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not B.Init(1, 1, ScreenWidth, ScreenHeight, MaxAvail) then begin
  WriteLn('Failed to init Browser. Status = ', InitStatus); Halt;
end;
```

Instantiates a full-screen browser with default window and color options using the largest contiguous free heap block for buffers.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt; HeapToUse : LongInt);
```

Purpose

Initialize a Browser with custom window options.

Description

InitCustom instantiates a Browser using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior of the window, where the file is displayed.

HeapToUse indicates the amount of maximum amount of memory that may be allocated for "pages." If the value specified is greater than MaxAvail, MaxAvail will be used as a limit instead. A "page" is an internal data structure with two components, a 4K buffer used to store a portion of the file being browsed and a small record used to keep track of the contents of the buffer. The Browser maintains a variably-sized array of these pages in an attempt to keep as much of the file in memory at once as possible. To minimize disk thrashing, a Least Recently Used (LRU) algorithm determines which page's buffer should be overwritten when new data has to be read in from disk.

InitCustom will fail if at least two pages cannot be allocated or if the parent constructor (CommandWindow.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Three sets of attributes in the specified ColorSet have special relevance for window objects of type Browser. BlockColor/BlockMono determine the attribute used to highlight marked blocks. MarkerColor/MarkerMono determine the attribute used to display text markers. And HighlightColor/HighlightMono determine the attribute used to highlight lines containing strings found by a search operation. After the object has been instantiated, these attributes may be changed by calling the SetBlockAttr, SetMarkerAttr and SetHighlightAttr methods, respectively.

Example

```
if not B.InitCustom(2, 2, ScreenWidth-1, ScreenHeight-1, DefaultColorSet,
                    DefWindowOptions or wBordered, MaxAvail) then begin
  WriteLn('Failed to init Browser. Status = ', InitStatus); Halt;
end;
```

Instantiates a bordered, full-screen browser with the default color set using the largest contiguous free heap block for buffers.

See Also

CommandWindow.InitCustom

Load / OpenFile

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a Browser from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Browser.Store. If the brStreamReload option was in effect when the Browser was stored, the stream will contain the name of the file that was loaded at the time. If there is such a filename, Load will attempt to open that file for browsing. If it is unable to do so, Load will fail, and InitStatus will contain an I/O error code reported by OpenFile.

The stream registration procedure for a Browser is BrowserStream.

Example

```
uses Dos, OpCrt, OpRoot, OpCmd, OpDos, OpFrame, OpWindow, OpBrowse;
var
  B : Browser;
  S : BufIdStream;
begin
  S.Init('BROWSE.STM', SOpen, 1024);
  S.RegisterHier(BrowserStream);
  S.Get(B);
  if S.GetStatus <> 0 then Halt;
  S.Done;
  B.Process;
  B.Erase;
  B.Done;
end.
```

In this example, we load B from a stream created by the Store example using the stream's Get method, which calls Browser.Load indirectly. Note that it is not necessary to call OpenFile here because the brStreamReload option was in effect when the object was stored.

See Also

CommandWindow.Load Store

Declaration

```
procedure OpenFile(FName : String);
```

Purpose

Open file for browsing.

Description

This routine attempts to open a file, FName, for browsing. Any I/O errors that occur while trying to open the file are reported to the user-defined error routine, if there is one. All I/O errors are classified as "fatal" by OpenFile. The only special error code returned by OpenFile is epFatal + ecFileIsEmpty, which indicates that the file exists but is completely empty.

Example

```
B.OpenFile('OPDEFINE.INC');
```

Loads OPDEFINE.INC into the browser.

See Also

CloseFile

Declaration
```
procedure Process; virtual;
```

Purpose
Process browse commands.

Description
The Browser's Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Browser Commands" at the start of this section.

Process calls the user-written status routine, if there is one, just prior to asking the user for the next command.

Example
See example at the beginning of this section.

See Also
CommandWindow.Process SetStatusProc

Declaration
```
procedure SetBlockAttr(Color, Mono : Byte);
```

Purpose
Set attributes for marked blocks.

Description
This method allows you to set the attribute used to highlight marked blocks.

Example
```
B.SetBlockAttr($4F, $70);
```
Displays marked blocks in white on red for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetDefaultExtension(DefExt : ExtStr);
```

Purpose
Default extension to use when prompting for filenames.

Description
This method sets the default extensions for filenames. ExtStr is a type ('string[3]') defined in OPSTRING.

Examples
```
B.SetDefaultExtension('PAS');
```
Sets the default file extension to 'PAS'.
```
B.SetDefaultExtension('.PAS');
```
Incorrect! Don't include the leading '.' in the extension.

See Also
SetGetFileProc

SetEditProc

Declaration

```
procedure SetEditProc(EF : BrowseEditFunc);
```

Purpose

Set edit function.

Description

This method allows you to designate a user-written routine that will be called when the Browser needs a string to be entered by the user. If such a routine is not present, certain commands will not be available (see the list of Browser Commands at the beginning of this section).

The routine should be of the form

```
{$F+}
function EditFunc(MsgCode : Word; Prompt : String;
                  ForceUp, TrimBlanks : Boolean;
                  MaxLen : Byte; var S : String) : Boolean;
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure.

MsgCode is the message code for the default Prompt. S is the string to be edited, and MaxLen is its maximum length. ForceUp indicates that the characters in the string should be forced to upper case. TrimBlanks indicates that leading and trailing spaces should be removed from the string. The function result should be True if the user entered anything, even if it is a blank string.

Example

```
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpBrowse, OpEdit;
var
  B : Browser;
  {$F+}
  function MyEditFunc(MsgCode : Word; Prompt : String;
                      ForceUp, TrimBlanks : Boolean;
                      MaxLen : Byte; var S : String) : Boolean;
  var
    LE : LineEditor;
  begin
    FastFill(80, ' ', 1, 1, TextAttr);
    LE.Init(DefaultColorSet);
    if ForceUp then
      LE.leEditOptionsOn(leForceUpper);
    if not TrimBlanks then
      LE.leEditOptionsOff(leTrimBlanks);
    LE.ReadString(Prompt, 1, 1, MaxLen, 40, S);
    MyEditFunc := (LE.GetLastCommand <> ccQuit);
    LE.Done;
    FastFill(80, ' ', 1, 1, TextAttr);
  end;
  {$F-}
...
B.SetEditProc(MyEditFunc);
```

Installs an edit function that displays prompts and accepts user input at the top of the screen. Note that the edit field width of 40 is somewhat arbitrary; in a real application the field width should be calculated based on the length of the prompt, MaxLen, and the space available.

See Also

SetGetFileProc

Declaration

```
procedure SetGetFileProc(GF : BrowseGetFileFunc);
```

Purpose

Set edit function.

Description

This method allows you to designate a user-written routine that will be called when the Browser needs a filename to be entered by the user. If such a routine is not present, certain commands will not be available (see the list of Browser Commands at the beginning of this section).

The routine should be of the form

```
{$F+}
function GetFile(MsgCode : Word; Prompt : string;
                 ForceUp, TrimBlanks, Writing, MustExist : Boolean;
                 MaxLen : Byte; DefExt : ExtStr;
                 var S : string) : Boolean;
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure.

MsgCode is the message code for the default Prompt. S is the string to be edited, and MaxLen is its maximum length. DefExt is the default extension to be applied to the filename, if any (ExtStr is declared in OPSTRING). ForceUp indicates that the characters in the string should be forced to upper case. TrimBlanks indicates that leading and trailing spaces should be removed from the string. If MustExist and Writing are both False, the user is being asked for the name of a new file that will be created later if necessary. If MustExist is True and Writing is False, the user is being asked for the name of an existing file that is to be merged into the current one. If Writing is True and MustExist is False, the user is being asked to enter the name of a file that will be overwritten if it already exists. The function result should be True only if the user entered the name of a file that meets the specified conditions.

Note that the default file extension, if there is one, will be applied to S by the calling routine if it hasn't been applied already by the user-written routine.

Example

```
uses
  OpCrt, OpDos, OpRoot, OpString, OpCmd, OpFrame, OpWindow, OpBrowse, OpEdit;
var
  B : Browser;
  {$F+}
  function MyGetFileFunc(MsgCode : Word; Prompt : string;
                         ForceUp, TrimBlanks, Writing, MustExist : Boolean;
                         MaxLen : Byte; DefExt : ExtStr;
                         var S : string) : Boolean;
  var
    LE : LineEditor;
    Cmd : Word;
  begin
    MyGetFileFunc := False;
    FastFill(80, ' ', 1, 1, TextAttr);
    LE.Init(DefaultColorSet);
    if ForceUp then
      LE.leEditOptionsOn(leForceUpper);
    if not TrimBlanks then
      LE.leEditOptionsOff(leTrimBlanks);
    LE.ReadString(Prompt, 1, 1, MaxLen, 40, S);
    Cmd := LE.GetLastCommand;
```

```
      LE.Done;
      FastFill(80, ' ', 1, 1, TextAttr);
      if Cmd = ccQuit then
        Exit;
      S := DefaultExtension(S, DefExt);
      if Writing then
        if ExistFile(S) then
          {** confirm that user wants to overwrite file **}
        else
          MyGetFileFunc := True
      else if ExistFile(S) or not MustExist then
        MyGetFileFunc := True
      else
        {** error -- file doesn't exist **} ;
    end;
    {$F-}
  ...
  B.SetGetFileProc(MyGetFileFunc);
```

This is a pretty complete example of an BrowseGetFileFunc. The chief things that are missing are 1) a calculated edit field width (40 is somewhat arbitrary), 2) a check to confirm that the user wants to overwrite a file (optional), and 3) an error message in the case where a file needs to exist and doesn't.

See Also

SetEditProc

Declaration
```
procedure SetHighlightAttr(Color, Mono : Byte);
```

Purpose
Set attributes for found text.

Description
This method allows to set the attribute used to highlight text found by a search operation.

Example
```
B.SetHighLightAttr($4F, $70);
```
Displays found strings in white on red for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetMarkerAttr(Color, Mono : Byte);
```

Purpose
Set attributes for text markers.

Description
This method lets you specify the attribute used to display text markers. If the attribute is $70, for example, and the current display is a monochrome monitor, then text marker 1 would be represented as a '1' in reverse video.

Example
```
B.SetMarkerAttr($2F, $70);
```
Displays place markers in white on green for color monitors, reverse video for monochrome monitors.

Declaration
```
procedure SetPrinter(LptNum : Byte);
```

Purpose
Set line printer (1-3).

Description
This method lets you designate the line printer used by the ccBlkPrint command. LptNum should be a value in the range 1-3, corresponding to LPT1-LPT3. If LptNum is not in this range, LPT1 will be used.

Example
```
B.SetPrinter(2);
```
Route printer output to LPT2.

SetStatusProc

Declaration

```
procedure SetStatusProc(SP : BrowseStatusProc);
```

Purpose

Set status procedure.

Description

This method allows you to specify a user-written routine that is to be called just before BrowseCommands.GetCommand is called to get the next editing command, and at various other times, notably when a potentially time-consuming operation such as a search operation begins. The routine should be of the form

```
{$F+}
procedure MyStatusProc(BP : BrowserPtr);
begin
end;
```

It must be compiled under the FAR model using {$F+}, and it may not be nested inside another procedure. The Browser making the call will always pass '@Self' as the BP parameter.

Be aware that OPBROWSE supplies a "built-in" status routine, called BrowseStatus, documented later in the chapter. You may use it as is, or you may use it as a model when writing your own.

Examples

```
uses
  Dos, OpCrt, OpString, OpRoot, OpCmd, OpDos, OpFrame, OpWindow, OpBrowse;
var
  B : Browser;
  {$F+}
  procedure MyStatusProc(BP : BrowserPtr);
  const
    StatusLine : string[37] = ' Line: xxxxx  Column: xxx  Filename: ';
  var
    S : string[5];
  begin
    with BP^ do begin
      S := Pad(Long2Str(brCurLine), 5);       {Line number}
      Move(S[1], StatusLine[8], 5);
      S := Pad(Long2Str(brColOfs+1), 3);      {Column at left edge}
      Move(S[1], StatusLine[23], 3);
      FastWrite(Pad(StatusLine+CurrentFileName, 80), 1, 1, TextAttr);
    end;
  end;
  {$F-}
...
B.SetStatusProc(MyStatusProc);
```

A simple status procedure that displays the current line, column, and file. Note that an actual status routine would insure that the filename did not exceed 43 characters.

```
B.SetStatusProc(BrowseStatus);
```

Activates OPBROWSE's built-in status procedure, which displays one line of status information on line StatusRow (default = 1) in the default attribute (white on black for both color and mono displays).

See Also

BrowseStatus

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a Browser in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Browser to the stream S. If the brStreamReload option is in effect, the stream will also contain the name of the file that is currently open for browsing. In that case, when Load is called to reload the object, OpenFile will be called to try to reopen the file.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a Browser is BrowserStream.

Example

```
uses
  Dos, OpCrt, OpRoot, OpCmd, OpDos, OpFrame, OpWindow, OpBrowse;
var
  B : Browser;
  S : BufIdStream;
  Status : Word;
begin
  if not B.InitCustom(2, 3, 78, 24,                {Window coordinates}
                      DefaultColorSet,             {ColorSet}
                      DefWindowOptions or wBordered, {Window options}
                      MaxAvail-8192)               {Max heap space to use}
  then begin
    WriteLn('Failed to init Browser. Status = ', InitStatus);
    Halt;
  end;
  B.OpenFile('BROWSE.PAS');
  B.brOptionsOn(brStreamReLoad);
  S.Init('BROWSE.STM', SCreate , 1024);
  S.RegisterHier(BrowserStream);
  S.Put(B);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  B.Done;
end.
```

Instantiates B, loads a file, and stores B in a stream using the stream's Put method, which calls Store indirectly. See Load for an example program that reloads this Browser.

See Also

CommandWindow.Store Load

ToggleHighBitStripping / ToggleModes / Transfer

Declaration

```
procedure ToggleHighBitStripping;
```

Purpose

Toggle stripping of high bits.

Description

This method allows you to activate or deactivate the stripping of high bits. Note that it must not be called until the browse window has been displayed once. If the window has not yet been displayed, 'brOptionsOn(brStripHigh)' or 'brOptionsOff(brStripHigh)' should be used instead.

Example

```
if B.brOptionsAreOn(brStripHigh) then
  B.ToggleHighBitStripping;
```

Forces high bit stripping off.

Declaration

```
procedure ToggleModes;
```

Purpose

Toggle between hex and ASCII modes.

Description

This method allows you to switch from ASCII to hex mode or vice versa. Note that it must not be called until the browse window has been displayed once. If the window has not yet been displayed, 'brOptionsOn(brHexMode)' or 'brOptionsOff(brHexMode)' should be used instead.

Example

```
if B.brOptionsAreOn(brHexMode) then
  B.ToggleModes;
```

Forces ASCII mode.

Declaration

```
procedure Transfer(StartPos, StopPos : LongInt; var Buffer; BufSize : Word);
```

Purpose

Copy browse characters from StartPos to StopPos into Buffer.

Description

This routine allows you to transfer data from the file being browsed into a buffer. One possible use for this routine might be in an application that displayed both a Browser and a TextEditor: Transfer could be used to copy the currently marked block into the editor.

StartPos and StopPos specify the range of data within the file that is to be transferred. Buffer is the destination for the data to be transferred. BufSize is the size of the buffer.

Example

```
{...check block markers, size of block, etc...}
B.Transfer(B.brBlkBegin.FilePos, B.brBlkEnd.FilePos, Buffer, BufSize);
{...check for error...}
```

Transfer data within marked block into a Buffer of BufSize bytes.

2. Procedures and Functions

This section documents ordinary procedures and functions that appear in OPBROWSE, as well as related type, constant, and variable declarations.

Constants

```
StatusRow   : Byte = 1;
StatusColor : Byte = $F;
StatusMono  : Byte = $F;
```

These typed constants affect the behavior of BrowseStatus. StatusRow is the screen row where the status line is displayed. StatusColor and StatusMono determine the video attribute in which the status line is displayed on color and monochrome displays, respectively.

BrowseStatus

Declaration

```
procedure BrowseStatus(BP : BrowserPtr);
```

Purpose

Display status line.

Description

This routine provides a ready-made status procedure for use with Browsers. The status line uses 80 columns, and is displayed at the row specified by the global typed constant StatusRow (defaults to 1) in either StatusColor or StatusMono, depending on the current display. The status line is of the form

```
FILENAME.EXT                    Working...              Line nnn    Col nnn
```

The first field shows the name of the file being browsed. The second field is displayed only when the browser is searching for a string or loading new data into its buffers. The third and fourth fields show the current line and column (refers to line at top of window).

Example

```
B.SetStatusProc(BrowseStatus);
```

Display status line on default row (1) in the default attribute (white on black for both color and mono displays).

```
StatusRow := ScreenHeight;
StatusColor := $0E;
StatusMono := $0F;
B.SetStatusProc(BrowseStatus);
```

Display status line at the bottom of the screen in either yellow on black (color displays) or white on black (mono displays).

See Also

Browser.SetStatusProc

E. Sample Programs

This section describes briefly three programs that demonstrate the use of the units covered in this chapter: MEMO (OPMEMO), EDITOR (OPEDITOR), and BROWSE (OPBROWSE).

1. MEMO

MEMO.PAS is a simple text editor based on OPMEMO. To use it, enter

```
MEMO filename
```

at the DOS prompt, where filename is the name of the file to be edited. If you do not specify a filename on the command line, you will be prompted for one.

In addition to the editing commands described in the documentation for OPMEMO, MEMO implements the following user-defined commands:

<CtrlK><S>, <F2>
Save file and continue editing.

<CtrlK><X>, <CtrlF2>
Save file and exit.

<CtrlK><Q>, <AltF2>
Abandon file. If the file has been modified, you will be asked to confirm that you want to quit without saving the changes. As implemented, this command is equivalent to ccQuit (<Esc>).

The top line of the screen is used to display messages, and the second line is used as a status line. The remainder of the screen is used as the edit window. Note that MEMO supports the EGA's 43-line mode and the VGA's 50-line mode.

This program is not really intended to be used, as EDITOR is, but rather to demonstrate how to use the MemoFile object (derived from Memo).

2. EDITOR

EDITOR is a more advanced text editor that has virtually all the features of the Turbo Pascal editor–and a few that it lacks, such as word wrap and mouse support. To use it, enter

```
EDITOR filename
```

at the DOS prompt, where filename is the name of the file to be edited. If you do not specify a filename on the command line, you will be prompted for one.

In addition to the editing commands described in the documentation for OPEDITOR, EDITOR implements the following user-defined commands:

<CtrlF2>
Save file and exit.

<AltF2>
Abandon file. If the file has been modified, you will be asked to confirm that you want to quit without saving the changes.

The top line of the screen is used to display messages and prompts, and the second line is used as a status line. The remaining lines are used as the edit window. Like MEMO, EDITOR supports the EGA's 43-line mode and the VGA's 50-line mode.

The last column on the screen displays a scroll bar. If a mouse is installed, you can use the scroll bar to move rapidly from one part of the file to another. If a mouse is not installed, the scroll bar simply shows the relative position of the cursor within the file.

3. BROWSE

BROWSE is a general purpose file browsing utility based on the OPBROWSE unit. To use it, enter

```
BROWSE filename.ext [filename.ext] [...] [filename.ext]
```

at the DOS prompt, where "filename.ext" is either the name of a file or a filemask containing wildcard characters. The following command lines are all valid, for example:

```
BROWSE browse.pas
BROWSE browse.pas opbrowse.pas
BROWSE *.pas
BROWSE *.pas *.inc
BROWSE *.pas *.inc *.asm
```

When multiple files are specified explicitly, or multiple files match a given filemask, you can go to the next file by pressing <Esc> or <ClickRight>. In the case of the last example shown above, for instance, BROWSE would first display all files with an extension of "PAS", then the "INC" files, then the "ASM" files.

Note that using the new file command–<F3> or <CtrlK><N>–does not affect BROWSE's behavior in this respect. For example, suppose you specified BROWSE.PAS and OPBROWSE.PAS on the command line, and while viewing BROWSE.PAS you used the new file command to load OPDEFINE.INC. If you were to press <Esc> while viewing OPDEFINE.INC, BROWSE would go ahead and load OPBROWSE.PAS, just as it would if you hadn't used the new file command.

In addition to the commands described in the documentation for OPBROWSE, BROWSE implements the following user-defined commands:

<F10>, <AltX>
Quit the program. This command will terminate the program even if there are more files to be browsed.

<AltH>
Toggle hex/ASCII mode.

The top line of the screen is used to display messages and prompts, and the second line is used as a status line. The remaining lines are used to display the file being browsed. Like MEMO and EDITOR, BROWSE supports the EGA's 43-line mode and the VGA's 50-line mode.

Also like EDITOR, BROWSE uses the last column on the screen to display a scroll bar. If a mouse is installed, you can use the scroll bar to move rapidly from one part of the file to another. If a mouse is not installed, the scroll bar simply shows the relative position of the line at the top within the file.

6. Data Entry

A. Overview

The size and scope of this chapter is admittedly somewhat intimidating. There's a lot of material here, and you shouldn't expect to digest it all in one sitting. When you boil it all down a bit, though, what you're left with are a few basic concepts, and two kinds of data entry facilities: those for editing individual variables one at a time (line editors), and those for editing groups of variables (entry screens). Line editors are used for tasks like prompting a user for the name of a file to be displayed. Entry screens are used for tasks like editing database records, which can have many component parts of varying types (string, integer, real, etc.).

But if you keep the pot boiling long enough, you can find the common denominators between them. By breaking the larger tasks performed by line editors and entry screens into a series of smaller tasks, you can develop an approach to data entry, a system, that should be adaptable to situations that weren't provided for initially. That was our primary goal in developing the units described in this chapter.

The basic building block of the system is the "field," an abstract concept that in **Object Professional** can mean different things in different contexts, and hence is difficult to describe. Perhaps the best starting point is a hierarchy of actual field types:

```
[DoubleListNode]
  AbstractField
    SelectField
      EntryField
        StringField
          ...
        MultiLineField
        [PickField]
    [PrintField]
      [StringPrintField]
        ...
      [LineField]
        [ShadedField]

[SingleListNode]
  TextField
```

In designing this hierarchy of objects, the main goal was to lay the groundwork for the higher level facilities in OPEDIT (line editing), OPSELECT (field selection), OPENTRY (screen editing) and OPFORM (printed forms). Each of these units contains an object that works with a list of one or more fields that are ultimately derived from an AbstractField.

An AbstractField has no data associated with it, only a set of methods that are needed by both EntryFields and PrintFields for formatting variables that are to be displayed on the screen or the printer. It is the common denominator between the data entry facilities described in this chapter and the printing facilities decribed in the next chapter.

If we follow the data entry branch of the tree, we arrive first at the SelectField, which stores the location, dimensions, and colors of a field. A SelectField is not associated with a particular variable, and hence cannot display itself; it merely describes a region of the screen and the colors in which a variable might be displayed. A Selector (OPSELECT) manages a list of SelectFields and provides methods for retrieving pre-formatted data and displaying it within the regions described by the SelectFields.

At the next level in the hierarchy, we find the EntryField, which has all the qualities of a SelectField, and then some. An EntryField *is* associated with a particular variable, and it knows how to convert the variable into a displayable form (a formatted string), display it, edit it, and validate a value assigned it by a user during editing. A LineEditor (OPEDIT) creates an EntryField to allow an individual variable to be edited. An EntryScreen (OPENTRY) creates a list of EntryFields and then manages the movement of the cursor from one field to another, allowing multiple variables to be edited as a group.

Although an EntryField is associated with a particular variable, it does not know the type of that variable. It is an abstract object, and you would never declare an instance of one. It simply defines data fields and methods that can be shared in common by fields of more specific types: StringField, WordField, RealField, and so on. Through the magic of procedure pointers and virtual methods, however, the numerous specific types can all be treated as a single type, an EntryField.

And therein lies the key to **Object Professional's** data entry system. It is set up in such a way that a higher level object, such as an EntryScreen, can treat all "fields" alike. When an EntryScreen needs to display the contents of a field, for example, it calls the field's Draw method, which will either perform the task itself or call a "driver" routine (whose address is stored in a procedure pointer) to do it. In preparation for that task, it will call the field's Convert method, which converts the corresponding variable from its native type into a string. When the field's contents need to be edited, the Edit method is called. Before allowing the user to leave the field, the editing routine will call the field's Validate method to determine whether or not the entered value is acceptable. When Edit is finished, the Convert method will be called a second time to convert the value represented by the string back to the appropriate type. In all these cases, the method in question will either perform the task itself or call the appropriate driver routine.

The field concept may be extended in many different directions. OPFIELD, for example, implements an object called the MultiLineField, which is used to display and edit an array of strings. OPENTRY defines a PickField object, which uses a PickList created with OPPICK to "edit" a string. And OPFORM extends the concept of the PrintField to embrace lines and shaded regions.

Returning to our diagram for a moment, we see that there is a second hierarchy of objects to be discussed here, one that is technically unrelated to the first. Although the TextField has some of the most basic qualities of an EntryField, it is a much simpler construct. A TextField just contains a string, the screen coordinates for the string, and a set of colors. Selectors and EntryScreens maintain singly-linked lists of these objects to store text that is to be displayed with the entry screen but never modified. You might have an entry screen that looks something like this, for example:

```
            Income      Expenses
January     11,070.51   4,753.19
February    13,679.23   5,214.08
March       12,206.12   4,927.16
April       14,561.17   5,598.67
```

The first column (with the names of the months) are probably prompts associated with the values in the second column. The third column is presumably a series of fields with no prompts. But the strings on the top row ('Income' and 'Expenses') are not obviously connected to any of the fields below them; they are best described as column headers. And this is what TextFields are for, displaying strings of text that are not associated with a particular field.

With this background information in mind, we can now turn our attention to the specific units discussed in this chapter. They fall into one of three categories: 1) units that contain low-level objects and routines that are used as building blocks, 2) units that implement high-level objects for performing data entry, and 3) related units.

The first category includes OPABSFLD, OPFIELD, and OPFEDIT. OPABSFLD defines the AbstractField object and a host of declarations relating to picture masks, which are used to format the contents of a field. OPFIELD defines the TextField, SelectField, and EntryField, as well as numerous field types derived from the EntryField. It also contains most of the driver routines for the various field objects. OPFEDIT contains a series of field editors, the driver routines used to edit fields of various types.

The second category consists of OPSELECT, OPENTRY, and OPEDIT. OPSELECT implements an abstract object called an AbstractSelector, from which both the Selector object in OPSELECT and the EntryScreen object in OPENTRY are derived:

```
[CommandWindow]
  AbstractSelector
    Selector
      ScrollingSelector
    EntryScreen
      ScrollingEntryScreen
```

The AbstractSelector defines the common ground between a Selector and an EntryScreen. It implements a number of low-level methods for working with fields, as well as methods that supply programming hooks useful for both Selectors and EntryScreens. The principal difference between a Selector and an EntryScreen is that an EntryScreen is more self-sufficient. A Selector simply allows the user to move the cursor from field to field and to select one of them, usually by pressing <Enter>; it then calls a routine that you must supply in order to edit the field. An EntryScreen can call the appropriate editing routine directly.

Like OPENTRY, OPEDIT provides an object, the LineEditor, which works with EntryFields, but it works with them one at a time. It differs from most other objects that process commands in that it is descended from Root rather than CommandWindow:

```
[Root]
  LineEditor
  SimpleLineEditor
```

In the final category are OPSEDIT and OPDATE. OPSEDIT provides a smaller, simpler alternative to OPEDIT's LineEditor, called the SimpleLineEditor. Instead, it implements a facility for editing strings (only) that is comparable in look and feel to the SimpleStringEditor in OPFEDIT, but it does not rely on OPABSFLD, OPFIELD, or OPFEDIT. Strictly speaking, OPDATE has nothing to do with data entry, but it is grouped with these units in this chapter because it is almost always used in conjunction with OPEDIT or OPENTRY. It provides routines for performing date and time arithmetic, and it is designed to work smoothly with the data entry facilities.

B. OPABSFLD: Abstract Field Definitions

Although you will only rarely need to use any of the code, data, or constants declared in OPABSFLD directly, this unit is one of the most important components of **Object Professional**, because it lays the groundwork for many of the higher level units. The low-level units OPFIELD and OPFEDIT both depend on it heavily, and the high-level units OPEDIT, OPSELECT, OPENTRY, and OPFORM all trace their roots to it. If you want to master those units, you must first grasp the concepts introduced in OPABSFLD.

The most important of these concepts is that of the "picture mask," a string of symbols, called "picture mask characters," that is used to describe both the legal contents of a "field" and the way that data associated with it is to be displayed. Section 1, below, describes picture masks in detail.

A related concept is that of the "character set." For each picture mask character defined in OPABSFLD, there is a corresponding character set that gives the mask character its meaning. They are described in section 2, below, along with the issue of user-defined picture mask characters.

In designing the data entry facilities in **Object Professional**, one of our chief goals was to develop a system that would make it possible not only for our overseas customers to write programs that conformed to the conventions of their respective countries, but indeed to allow the writing of programs that could adapt themselves to different conventions at runtime. This issue is discussed in section 3.

Finally, OPABSFLD defines an object called the AbstractField, the main purpose of which is to allow OPFIELD/OPENTRY and OPFORM to share code. The AbstractField has no data associated with it, only methods relating to picture masks. It is discussed briefly in section 4.

1. Picture Masks

As indicated earlier, picture masks serve two purposes. On one hand, they are used to control the formatting of data. You can use them to ask, for example, that a date be displayed in 'mm/dd/yyyy' format, meaning that the month should be displayed first, followed by the day and then the year, with a slash separating each component of the date from the other: '12/31/1999'. In this sense, the picture mask controls the process by which a value is converted from its native type (Integer, Real, Extended, Date, etc.) into a formatted string.

In the case of data entry fields, however, picture masks serve a secondary purpose, that of limiting the range of characters that may be entered at a given position in a string. In this sense, a picture mask provides the means by which a field editor may perform input validation on a character-by-character basis. More stringent tests may need to be performed later, after the user has accepted the contents of a field, but there's no point in performing them at all if the user has entered 'ab/cd/efgh' in a field that expects a date. A field editor should be able to reject characters that clearly make no sense at the time they are entered, and with the help of a picture mask it can.

Before going any farther, we need to introduce another basic concept. Consider again the example of the picture mask for a date field, 'mm/dd/yyyy'. This field will actually have three component parts--a month, a day, and a year. These individual parts are referred to as "subfields." Many seemingly simple data types actually break down into multiple parts when they are displayed on the screen: dates, times (hours, minutes, seconds), phone numbers,

zip/postal codes, even real numbers. The characters used to separate one subfield from another are called, aptly, "separators." The general form of a field is thus something like this:

```
Subfield ([Separator] [Subfield]) ... ([Separator] [Subfield])
```

In most cases, a field consists of a single subfield, but you should keep in mind while reading this section that matters are not always so simple. We'll return to the issue of subfields and separators a bit later.

A picture mask may consist of one or more of three classes of mask characters: formatting characters, literal characters, and semi-literal characters.

As far as data entry is concerned, formatting characters denote those places in a field where new text may be entered by the user or existing text may be modified. Each formatting character is associated with its own character set (see section 2, below), and only characters within that set may be entered in the corresponding slot. For example, the picture mask '999' indicates that only numbers ('0'..'9') and spaces (' ') may be entered in a given field. Several formatting characters--'!', 'A', and 'L', for example--also indicate that input should automatically be converted to upper or lower case. As far as data conversion for display purposes is concerned, formatting characters denote places where a field's "raw" value may be plugged into the picture mask. For example, given the picture mask '$999.99' and a real value of 1.201, the result of merging the value with the picture mask would be ' $1.20'. The 9's indicate the locations in the output string where the digits to the left and right of the fixed decimal point should appear.

In general, any character in a picture mask that has not been assigned a function as a formatting character is treated as a literal, meaning that location in the field may not be altered during editing, and that it will not be altered when a value is converted to a string. For example, given a picture mask of '[999]' and any integer value between 0 and 999, the result of merging the value with the picture mask will always be a string that starts with '[' and ends with ']', because both characters are treated as literals. In the case of data entry fields of type string, these literal characters become a permanent part of the string being edited, to insure that they will look the same way when you display them yourself as they do when they are being edited. In most other cases, the literal characters in a picture mask are stripped out in the process of converting a string back to its native data type, and restored when converting it back to a string.

Semi-literal characters are like literal characters in the sense that they usually correspond to locations in a field that may not be altered while editing. The two currency characters ('c' and 'C'), for example, always mark slots in a field that cannot be changed. What appears on the screen and the edited string, however, is usually a '$' (this can be changed). In a couple of cases, however, a semi-literal character requires more than a simple substitution. The floating dollar sign ('$') sets aside a slot in a field that will be used only if the rest of the field is full. If the field is not full, the dollar sign will be moved just to the left of the leftmost digit. Somewhat similarly, the comma character (',') indicates a slot that is to be filled with a comma (or the value of CommaChar) only if there is a digit to put to the left of it. Otherwise its slot is filled with a dollar sign, minus sign, or space. In neither case, however, will a slot set aside for a semi-literal character be filled with a digit entered by the user.

Before this gets too abstract, let's look at the picture mask characters defined by **Object Professional**, many of which will be familiar to dBase programmers. We've arranged them into four categories, the last of which covers picture mask characters that are actually defined in OPDATE rather than OPABSFLD. In each case, what is shown is an actual constant

declaration from either OPABSFLD or OPDATE, followed by a brief description of what the mask character represents.

```
Regular Mask Characters
-----------------------------------------------------------------------------
AnyChar       = 'X';              {any character}
ForceUp       = '!';              {any character, force upper case}
ForceLo       = 'L';              {any character, force lower case}
ForceMixed    = 'x';              {any character, force mixed case (entry only)}
AlphaOnly     = 'a';              {alphas only}
UpperAlpha    = 'A';              {alphas only, force upper case}
LowerAlpha    = 'l';              {alphas only, force lower case}
NumberOnly    = '9';              {numbers and spaces only}
DigitOnly     = '#';              {numbers, spaces, minus, period}
Scientific    = 'E';              {numbers, spaces, minus, period, 'e'}
HexOnly       = 'K';              {0-9 and A-F, force upper case}
BooleanOnly   = 'B';              {T, t, F, f}
YesNoOnly     = 'Y';              {Y, y, N, n}

User-Defined Mask Characters
-----------------------------------------------------------------------------
User1         = '1';              {for user-defined character sets 1..8}
...
User8         = '8';

Semi-Literal Mask Characters
-----------------------------------------------------------------------------
FloatDollar   = '$';              {floating dollar sign}
CurrencyLt    = 'c';              {currency to left of the amount}
CurrencyLtStr : string[5] = '$'; {corresponding string}
CurrencyRt    = 'C';              {currency to right of the amount}
CurrencyRtStr : string[5] = '';  {corresponding string}
DecimalPt     = '.';              {insert decimal point}
DecimalChar   : Char = '.';       {corresponding character}
Comma         = ',';              {character used to separate numbers}
CommaChar     : Char = ',';       {corresponding Character}

Mask Characters in OPDATE
-----------------------------------------------------------------------------
MonthOnly     = 'm';              {month - zero-padded}
MonthOnlyU    = 'M';              {month - blank-padded}
DayOnly       = 'd';              {day - zero-padded}
DayOnlyU      = 'D';              {day - blank-padded}
YearOnly      = 'y';              {year - zero-padded}
NameOnly      = 'n';              {month name}
NameOnlyU     = 'N';              {month name - force upper case}
WeekDayOnly   = 'w';              {weekday}
WeekDayOnlyU  = 'W';              {weekday - force upper case}
DateSlash     = '/';              {mm/dd/yy separator (semi-literal)}
HourOnly      = 'h';              {hours - zero-padded}
HourOnlyU     = 'H';              {hours - blank-padded}
MinOnly       = 'm';              {minutes - zero-padded}
MinOnlyU      = 'M';              {minutes - blank-padded}
SecOnly       = 's';              {seconds - zero-padded}
SecOnlyU      = 'S';              {seconds - blank-padded}
TimeOnly      = 't';              {'p'/'a' in 'am'/'pm'}
EmOnly        = 'e';              {'m' in 'am'/'pm'}
TimeColon     = ':';              {hh:mm:ss separator (semi-literal)}
```

Those in the first group, the "regular" mask characters, are the ones most commonly used. These are used to format strings, numbers, characters, and boolean values. Each is associated with a set of characters that, during data entry, determines what characters may be entered at a given position in a string, and each is associated with a case-changing quality: no change, force upper case, force lower case, or force mixed case. Unless the comment on the right indicates otherwise, the case-changing quality of the mask character is "no change."

Three of the regular mask characters deserve special attention. The 'x' character, which forces "mixed case," affects only the entry of new data. If the user enters 'sam jones' in an empty field, for example, the result will be 'Sam Jones'. That is, if the cursor is at the first position in a subfield, or the previous character is a space, the next character entered will be converted to upper case. Characters in other positions are not forced to upper case, nor is the case of existing text changed when a field is displayed or when insertion/deletion of characters alters the position of other characters.

The 'E' character, which allows real numbers to be entered/displayed using scientific notation, may be.used *only* with "simple" field types. See the documentation for OPFIELD for a discussion of "simple" fields.

The 'K' character, which allows numbers to be entered/displayed in hexadecimal format should generally be used only with fields of an integer type: Byte, ShortInt, Integer, Word, and LongInt. Furthermore, if a picture mask contains even one instance of 'K', it is assumed that there is a single subfield in the picture mask consisting entirely of 'K' characters. This assumption is made to allow a single conversion routine (discussed in the OPFIELD section) to handle numbers entered in either decimal or hexadecimal format.

Those in the second group, the "user-defined" mask characters, are set aside for your personal use. They provide a means of defining new mask characters to meet special needs. Examples of how to use them are given in the next section, on "Character Sets and User-Defined Mask Characters."

The third group consists of both semi-literal mask characters and their corresponding values (the typed constants). In the simplest cases, the mask character represents a location in the field where a one-to-one substitution is made when the field is being formatted. If CommaChar = '.', for example, then all instances of ',' in the picture mask will be translated to '.' when the field's value is merged with the picture mask. This subject is discussed in greater detail in the section on "International Currencies and Conventions," below.

The fourth group consists of formatting characters and semi-literal characters defined in OPDATE. These are discussed in the documentation for that unit.

To give you a better idea of what the mask characters mean, here are examples of some typical picture masks and some resulting strings:

```
Picture          Result          Remarks
-------------------------------------------------------------
'XXXXXXXXXXX'  'Hello world!'  any character accepted
'!!!!!!!!!!!'  'HELLO WORLD!'  any character, force upper case
'LLLLLLLLLLL'  'hello world!'  any character, force lower case
'xxxxxxxxxxx'  'Hello World!'  any character, force mixed case
                               (on entry only)
'aaaaaaaaaaa'  'Hello world '  alphas only
'AAAAAAAAAAA'  'HELLO WORLD '  alphas only, force upper case
'lllllllllll'  'hello world '  alphas only, force lower case
'[9999]'       '[ 123]'        digits and ' ' only, brackets are literal
```

Picture	Result	Remarks
'[####]'	'[-123]'	'-' is allowed
'EEEEEEEEEEE'	'1.234567E+17'	scientific notation
'KKKK'	'07FE'	hexadecimal numbers
'B'	'T'	booleans (True-False)
'Y'	'N'	yes-no
'c999.99'	'$ 12.50	dollar sign is fixed
'$999.99'	' $12.50'	dollar sign floats
'999.99CC'	'123,45 F'	French francs, fixed (DecimalChar = ',')
'$###,###.##'	' $1,234.56'	',' is discretionary, '$' floats
'$###,###.##'	' $12.34'	
'mm/dd/yyyy'	'01/16/1926'	month is padded with 0's
'MM/dd/yyyy'	' 1/16/1926'	month is padded with spaces
'mm/dd/yy'	'01/16/26'	
'MM/dd/yy'	' 1/16/26'	
'dd/mm/yyyy'	'16-01-1926'	British date format (SlashChar = '-')
'dd-nnn-yyyy'	'16-Jan-1926'	month name instead of number (SlashChar = '-')
'hh:mm'	'23:30'	time in military format
'hh:mmt'	'11:30p'	time in am/pm format, no 'm'
'HH:mm te'	' 1:30 pm'	am/pm format, hour padded with spaces

You may have noticed in the examples given here that there is no mixing of formatting characters within subfields. For instance, there are no examples of masks like "!XXXX" (convert first character to upper case, leave others alone). Although picture masks like these are legal, they are not guaranteed to produce the desired results, especially in the case of data entry fields. Why? Because there are simply too many situations in which the rules seemingly imposed by the picture mask become ambiguous. For example, what if the picture mask were (nonsensically) '9#LA'? It would be easy enough to enforce the implied restrictions if the field were empty initially and new text was being entered, but what should be done if the field was full ('1.uA') and the user tried to delete the first character in the string, thereby pulling the end of the string toward the front ('.uA ')? Should we disallow the deletion? What if the user wanted only to delete the character and immediately insert another, valid character in its place? The approach taken in **Object Professional** is to enforce the rules implied by the picture mask only at the time a given character is being entered. It is more prudent in such cases, we think, to let the user enter what he wants and validate the entry by some other means later on.

One other thing you should notice in these examples is that many of them represent fields that are comprised of multiple parts, or subfields. And in each instance, the separator is either a literal or a semi-literal character, never a formatting character (such as '9' or '!'). In terms of picture masks, a subfield is generally defined as a series of formatting characters bounded by either a subfield separator (such as '/' or '-'), a literal character (such as '['), a semi-literal character with a fixed position (such as 'c'), or one end of the field. The only exceptions to this rule are the rare cases where nothing separates two subfields, such as 'hh:mmt', where the 't' immediately follows the final 'm' in the minutes subfield. Although these cases can be handled if necessary, they should generally be avoided for much the same reason that mixing of formatting characters within a subfield should be avoided.

2. Character Sets and User-Defined Mask Characters

Although the concept of the character set is intimately connected to that of the picture mask, we have separated the two because it is not always necessary to know the name or the exact contents of a given character set in order to use the picture mask character associated with it. We also wanted to focus particular attention on user-defined picture mask characters, which you simply cannot use without understanding character sets.

As we have said before, character sets define the limits that are imposed by picture mask characters. If, for example, the first character in a field's picture mask is '9', that means that only digits ('0'..'9') and spaces may be entered at the first position in the field. That meaning is not written in stone, however. It is a function of the fact that the corresponding character set, NumberOnlySet, is defined thus:

```
const
  NumberOnlySet : CharSet = ['0'..'9', ' '];
```

As you can see, the character set is a typed constant, which means that it can be changed at runtime if desired. We wouldn't want to change it in this case, but in others we might need to change it in order to achieve a desired effect. The most notable cases are those involving the character sets associated with user-defined picture mask characters. (Two other important cases will be discussed in the next section.)

A typical application of a user-defined picture mask character ('1'..'8') might be to limit entry in a character field to a narrow range of non-contiguous values. Although there are other ways you could validate entries into such fields, this is usually the easiest and most efficient. For example:

```
UserSet1 := ['D', 'R', 'I'];       {Democrat, Republican, Independent}
ForceCaseUser['1'] := UpperCase;
UserSet2 := ['m', 'f'];            {Male or Female, forced to lower case}
ForceCaseUser['2'] := LowerCase;
UserSet3 := ['M', 'm', 'F', 'f']; {Male or Female}
ForceCaseUser['3'] := NoChange;
UserSet4 := ['A'..'N', ' '];       {alphas A..N, blank}
ForceCaseUser['4'] := UpperCase;
```

Given these statements, a picture mask of '1' would allow only 'D', 'R', and 'I' to be entered into the field and input would be forced to upper case, while a picture mask of '2' would allow only 'm' and 'f' and input would be forced to lower case. The example for '3' is similar to that for '2', except that the case of input characters would not be changed. A picture mask of '4' would allow only the alphabetic characters 'A' through 'N' or a blank space; input would be forced to upper case. In all these cases, the same two steps are required in order to define the meaning of the mask character UserN: 1) you initialize the corresponding character set, UserSetN, with the desired range of values and 2) you initialize ForceCaseUser[N] to define the case changing characteristics of the mask character.

The concept is equally useful for string fields. Say, for example, that you wanted to define a picture mask character that had the same meaning as AlphaOnly ('a'), but forced mixed case rather than upper ('A') or lower ('l'), just as ForceMixed ('x') does. Here is what you would do:

```
UserSet5 := AlphaOnlySet;
ForceCaseUser['5'] := MixedCase;
```

You could then create a mask like '555555555' that would impose the same limits on characters entered as 'aaaaaaaaa', but with the case changing characteristics of 'xxxxxxxxx'.

Below are the declarations for all the character sets defined in OPABSFLD, as well as the types and constants relating to case change:

Types

```
CaseChange = (NoChange, UpperCase, LowerCase, MixedCase);
```

The different types of case change operations that can be associated with a picture mask character. NoChange means that the case of a character is left alone. UpperCase and LowerCase mean that the character will be converted to upper or lower case, respectively. MixedCase means that, during data entry, a character will be forced to upper case if the cursor is at the beginning of a subfield or if the previous character is a space.

Constants

```
ForceCaseUser : array[User1..User8] of CaseChange = (
  NoChange, NoChange, NoChange, NoChange,
  NoChange, NoChange, NoChange, NoChange);
```

The values in this array specify the case changing characteristics of the user-defined picture mask characters User1..User8 ('1'..'8').

```
AnyCharSet    : CharSet = [#0..#255];                {'X', '!', 'L', 'x'}
AlphaOnlySet  : CharSet =                            {'a', 'A', 'l'}
  ['A'..'Z', 'a'..'z', #128..#154,
  #160..#167, ' ', '-', '.', ','];
NumberOnlySet : CharSet = ['0'..'9', ' '];           {'9'}
DigitOnlySet  : CharSet = ['0'..'9', ' ', '-', '.']; {'#'}
ScientificSet : CharSet =                            {'E'}
  ['0'..'9', ' ', '-', '+', '.', 'e', 'E'];
HexOnlySet    : CharSet =                            {'K'}
  ['0'..'9', ' ', 'A'..'F', 'a'..'f'];
BooleanSet    : CharSet = ['T','t','F','f'];         {'B'}
YesNoSet      : CharSet = ['Y','y','N','n'];         {'Y'}
UserSet1      : CharSet = [#0..#255];                {'1'}
UserSet2      : CharSet = [#0..#255];                {'2'}
UserSet3      : CharSet = [#0..#255];                {'3'}
UserSet4      : CharSet = [#0..#255];                {'4'}
UserSet5      : CharSet = [#0..#255];                {'5'}
UserSet6      : CharSet = [#0..#255];                {'6'}
UserSet7      : CharSet = [#0..#255];                {'7'}
UserSet8      : CharSet = [#0..#255];                {'8'}
```

These are the character sets corresponding to all of the formatting characters defined in OPABSFLD. The comment on the right shows the mask character(s) associated with each character set.

3. International Currencies and Conventions

In adapting a program written with **Object Professional** for use in a country other than the United States, there are really only a few areas of concern. First and most obvious are the language differences, if any. OPROOT supplies an answer for part of this problem by defining codes for all messages generated by **Object Professional**. A similar issue arises with boolean and yes-no fields, since 'T'/'F' and 'Y'/'N' aren't appropriate in all countries. To deal with this particular problem, you need to do two things: change the contents of the character sets BooleanSet and YesNoSet, shown in the previous section, and change the following typed constants:

```
TrueChar  : Char = 'T';
FalseChar : Char = 'F';
YesChar   : Char = 'Y';
NoChar    : Char = 'N';
```

A program written for use in a French-speaking country or province, for example, would need to change 'Y' (Yes) to 'O' (Oui):

```
YesNoSet := ['O','o','N','n']; {Oui/Non}
YesChar := 'O';                {Oui}
NoChar := 'N';                 {Non}
```

The first statement defines the characters that may be entered in a yes-no field, while the second affects the way that a value of True (= Yes) is displayed on the screen.

In addition to the language differences, different countries have their own conventions on how dates and times and monetary values are represented. Most European countries, for example, put the day before the month ('31/12/1999'), whereas in the United States the order is reversed ('12/31/1999'). And the character used to separate the various parts of the date can vary too. In Switzerland, they use a period rather than a slash or a dash: '31.12.1999'.

Picture masks make it simple to deal with differences in the order of things. If you want the day to be displayed before the month, you simply change the order of the subfields in the picture mask. Differences in "separators" are handled by means of the semi-literal mask characters, which correspond to typed constants whose values may be changed. These are the semi-literal characters:

Char	Meaning	Corresponding constant
'.'	decimal point	OpAbsFld.DecimalChar
','	thousands separator	OpAbsFld.CommaChar
'$'	floating currency	OpAbsFld.CurrencyLtStr
'c'	fixed currency - left	OpAbsFld.CurrencyLtStr
'C'	fixed currency - right	OpAbsFld.CurrencyRtStr
':'	time separator	OpDate.ColonChar
'/'	date separator	OpDate.SlashChar

To return to the case of dates in Switzerland: we would solve that problem by using a picture mask of 'dd/mm/yyyy' and setting SlashChar to '.'.

Since the subject of international dates and times is covered in detail in the documentation for OPDATE, let's focus on the issue of international currencies, which is addressed in a similar manner. Suppose that you need to construct a picture mask for a field that will contain a monetary value in the range 0.00 to 999,999.99. Here are a few examples of how you would configure **Object Professional** to deal with a given country's currency:

```
USA
-------------------------------------------------------------
CurrencyLtStr = '$';                    {default}
CurrencyRtStr = '';                     {default}
DecimalChar   = '.';                    {default}
CommaChar     = ',';                    {default}
PictureMask1  = '$999,999.99';          {floating currency symbol}
Result        = '  $1,234.56'
PictureMask2  = 'c999,999.99';          {fixed currency symbol}
Result        = '$ 12,345.67'

Germany
-------------------------------------------------------------
CurrencyLtStr = 'DM';
CurrencyRtStr = '';
DecimalChar   = ',';
CommaChar     = '.';
PictureMask1  = '$$999,999.99';
Result        = '  DM1.234,56'
PictureMask2  = 'cc999,999.99';
Result        = 'DM 12.345,67'

France
-------------------------------------------------------------
CurrencyLtStr = '';
CurrencyRtStr = ' F';
DecimalChar   = ',';
CommaChar     = ' ';
PictureMask2  = '999,999.99CC';
Result        = ' 12 345,67 F'
```

As you can see, **Object Professional** can adapt to currency symbols of varying length (1-5 characters) on either side of a number. The key to getting the proper output is to ensure that the picture mask is coordinated with the values of CurrencyLtStr and CurrencyRtStr. If the currency symbol is a single character on the left, then CurrencyLtStr should contain that character and a single 'c' or '$' should appear at the beginning of the picture mask. If it is a two-character string on the right, then CurrencyRtStr should contain those two characters and 'CC' should appear at the end of the picture mask.

How do you ensure that CurrencyLtStr, CurrencyRtStr, DecimalChar, and CommaChar are initialized properly and that your picture masks reflect their values? In most cases, you simply supply values that you know are correct for your own country (or by leaving the default values alone if they are correct) and by designing your picture masks accordingly. In cases where your program must adapt to the conventions of different countries, however, you'll need some help. This help comes in the form of the InternationalCurrency routine supplied by OPABSFLD and documented below.

Important: If you do not use InternationalCurrency and you need to change DecimalChar, you should use the ChangeDecimalChar procedure described below. Besides changing DecimalChar, it also fixes the two character sets affected by the change, DigitOnlySet and ScientificSet. (InternationalCurrency does this as well.)

Declaration

```
procedure ChangeDecimalChar(DC : Char);
```

Purpose

Reset DecimalChar to DC.

Description

This routine changes the global variable DecimalChar to DC. More importantly, it also changes DigitOnlySet and ScientificSet so that they are in agreement with DecimalChar. This step is necessary to insure that line editors and data entry screens recognize the alternate character as a decimal point.

Example

```
ChangeDecimalChar(',');
CommaChar := '.';
```

Change DecimalChar to ',' and CommaChar to '.' (appropriate for Germany).

Declaration

```
function InternationalCurrency(FormChar : Char; MaxDigits : Byte;
                               FloatIfPossible, AddCommas : Boolean) : string;
```

Purpose

Return a picture mask for a currency string, based on DOS's country info.

Description

This routine uses DOS's Get Country Information service to determine not only the proper values for DecimalChar, CommaChar, CurrencyLtStr, and CurrencyRtStr, but also how to produce a picture mask that is appropriate for the current country's currency.

FormChar is the formatting character to use (usually '9' or '#'). MaxDigits is the number of digits that may appear to the left of the decimal point (DOS indicates how many there should be to the right). FloatIfPossible indicates whether '$' or 'c' should be used if the currency symbol goes on the left. AddCommas indicates whether discretionary commas should appear in the picture mask.

When using this routine, you need to keep in mind that the string returned might be as large as 8+MaxDigits characters long, not counting any space set aside for commas (Pred(MaxDigits) div 3 characters). This is because a currency symbol can be as long as 5 characters, and there will usually be a decimal point with two digits on its right as well. You should also bear this in mind when laying out your entry screens: leave room for the largest picture mask that you might possibly receive.

Note that the information provided by DOS 2.x is minimal compared to that available under DOS 3.x and higher. We therefore recommend that this function be used with caution in programs that must run under DOS 2. Where information is lacking, US conventions are followed.

Examples

```
InternationalCurrency('9', 6, True, True)
```

Returns '$999,999.99' for US (DecimalChar = '.', CommaChar = ',', CurrencyLtStr = '$', CurrencyRtStr = ''), '999,999CCCC' for Italy (DecimalChar unused, CommaChar = '.', CurrencyLtStr = '', CurrencyRtStr = 'Lit.').

```
InternationalCurrency('#', 4, False, False)
```

Returns 'c####.##' for US, 'cccc####.##' for Sweden (DecimalChar = ',', CommaChar = '.', CurrencyLtStr = 'SEK ', CurrencyRtStr = '').

See Also

InternationalDate (OPDATE) InternationalTime (OPDATE)

4. The AbstractField

Like several other low-level objects in **Object Professional**, the AbstractField exists only to be overridden by higher level ones. More specifically, it exists to allow the print fields in OPFORM and the data entry fields in OPFIELD to inherit methods needed to work with picture masks. You will never need to declare an instance of an AbstractField, because it doesn't do anything useful in and of itself. It does, however, declare some methods that you may need to use at some point.

Because the methods in AbstractField are so rarely called directly, we will depart a bit from the standard format of the manual here. Below are the relevant type declarations in OPABSFLD, followed by brief comments on what the more important methods do.

Types

```
PictureFlags  = array[1..255] of Boolean;
AbstractField =
  object(DoubleListNode)
    procedure InitPictureFlags(var PFlags : PictureFlags);
      {-Initialize a picture flags table}
    procedure MergePicture(St : string; var S : string);
      {-Merge St with Picture and return result in S}
    procedure StripPicture(var S1, S2 : string);
      {-Strip picture characters out of S1 and return in S2}
    procedure CalcWidthAndPlaces(var Width, Places : Word);
      {-Calculate width and decimal places for a numeric field}

    {+++ internal methods +++}
    function afMaxLen : Byte; virtual;
    function afPicture : String; virtual;
    function afDPlaces : Byte; virtual;
    function afNoLiterals : Boolean; virtual;
    function afNumeric : Boolean; virtual;
    function afHexadecimal : Boolean; virtual;
    function afRightJustified : Boolean; virtual;
    function afIsNumber : Boolean;
  end;
```

The internal methods with the 'af' prefix are ones that you should never call directly. In general, these are just hooks that allow other methods (such as MergePicture) to obtain information that isn't available until the internal methods (such as afPicture, which returns the picture mask for the field) have been overridden by a derived object.

InitPictureFlags is used to initialize a PictureFlags array based on the contents of a picture mask. Its job is to construct an array of booleans that indicates those positions in a field where a user may enter text.

MergePicture takes a raw, unformatted string representing a field's value and merges it with the picture mask. This is how '1.234' gets combined with '$999.99' to form ' $1.23'. It is used by conversion routines when formatting a field before it is displayed.

StripPicture reverses the process performed by MergePicture. It is used by conversion routines after a field has been edited to extract the relevant data from a formatted string.

CalcWidthAndPlaces is used by the conversion routines for real numbers to calculate the values to be passed to Str when turning the real into a raw string ('Str(RealVar:Width:Places, StringVar)').

C. OPFIELD: Data Entry Fields

This section is divided into 8 parts. Part 1 deals with the TextField, an object used by Selectors (OPSELECT) and EntryScreens (OPENTRY) to create display-only fields. Part 2 discusses the SelectField, an object used by Selectors to store information about the location, size, and appearance of a field. Part 3 covers the abstract EntryField, which is derived from the SelectField and is used by EntryScreens to store all relevant information about a field within a data entry screen. Part 4 covers most of the type-specific objects derived from an EntryField: StringField, WordField, RealField, etc.

The next three parts cover three of the four categories of driver routines used by the various objects derived from EntryField: conversion routines, validation routines, and drawing routines. The fourth category of driver routine, field editors, is discussed in the section on OPFEDIT.

Part 8 discusses the stream registration routines for the various field types, a subject that you'll need to know about only if you plan on working with entry screens stored in streams.

Important: A special note concerning all these sections is in order here. In contrast to our practice in most other sections of the manual, the methods and routines discussed here are documented either in groups or as part of a general discussion of a particular object. There are several reasons for giving them this special treatment.

First, it avoids an awful lot of redundancy. All of the conversion routines take the same parameters, for example, as do all the validation routines and all the registration routines. It would be a waste of valuable space to document these routines individually one- or two-to-a-page. Once you know how to call one registration routine, you know how to call all of them. The same is generally true of many of the field objects. Most of them look alike externally, and a few words on their internal differences is generally enough to distinguish them.

Beyond that, it is also fair to say that you can do a lot with Selectors and EntryScreens without knowing any of the intimate details about the field objects they work with. The main reason for documenting them at all is to aid the more hardy among you in the development of derived field types and/or the development of objects derived from Selectors or EntryScreens. If you're not developing derived types, there are very few details that you need to know about these lower level objects.

1. The TextField

As indicated in the introduction to this chapter, TextFields are used to create display-only fields in Selectors and EntryScreens. Unlike SelectFields and EntryFields, you will almost never need to work with a TextField directly. They are typically added to a Selector/EntryScreen by calling one of the AddTextField methods, which instantiate them and then add them to a singly-linked list stored in the larger object, and then left alone. The Selector/EntryScreen handles the task of displaying them automatically.

What follows is a discussion of all type and constant declarations pertaining primarily or exclusively to TextFields.

Types

```
TextFieldPtr = ^TextField;
TextField =
  object(SingleListNode)
    tfID           : Word;      {field ID--assigned automatically}
    tfFlags        : Word;      {flags--currently unused}
    tfRow, tfCol   : Word;      {coordinates for the text}
    tfColorAttrs   : FlexAttrs; {colors}
    tfMonoAttrs    : FlexAttrs;
    tfString       : StringPtr; {text string}

    constructor Init(FieldID : Word; var St : String; Row, Col : Word;
                     var ColorAttrs, MonoAttrs : FlexAttrs);
      {-Called when field is created}
    destructor Done; virtual;
      {-Called when field is destroyed}

    procedure Draw(XDelta, YDelta : Byte); virtual;
      {-Draw the field}

    constructor Load(var S : IdStream);
      {-Load a text field from a stream}
    procedure Store(var S : IdStream);
      {-Store a text field in a stream}
  end;
```

The tfID field stores the ID number for the field. This value is assigned automatically by the AddTextField method in a Selector or EntryScreen to allow a particular text field to be located later using FindTextField. (The first TextField added to the list is assigned an ID of 0, the next is 1, and so on.) tfFlags is reserved for future internal use. tfRow and tfCol store the relative coordinates for the field. tfColorAttrs and tfMonoAttrs store the video attributes for the field. Unlike other field types, a TextField is always displayed using the FlexWrite routine in OPCRT, rather than the FastWrite routine. Hence the attributes are stored in a pair of FlexAttrs arrays, rather than a pair of bytes. tfString points to the string of text associated with the field.

Init is the only constructor for the object other than Load. Apart from storing the values passed to it as parameters, the only thing it does is to to allocate a copy of the St parameter on the heap. This string is deallocated by the Done destructor.

The Draw method displays the string pointed to by tfString using FlexWrite. The parameters passed to Draw indicate values to be added to tfRow and tfCol before displaying the field. If the TextField is connected to a non-scrollable Selector or EntryScreen, these values allow the relative coordinates for the field to be converted to absolute coordinates: tfRow + YDelta is the absolute row, and tfCol + XDelta is the absolute column. If the TextField is connected to a ScrollingSelector or a ScrollingEntryScreen, the parameters passed will always be 0, because the coordinates of the field are relative to the top left corner of the virtual screen on which all the fields are drawn before the appropriate section of the virtual screen is copied to the virtual screen. (This matter is discussed more fully in the sections on OPSELECT and OPENTRY.)

Load and Store are needed only when a Selector or EntryScreen is being stored in or retrieved from a stream. You will never need to call them directly.

2. The SelectField

As indicated in the introduction to the chapter, the SelectField serves two main purposes. On the one hand, a doubly-linked list list of SelectFields can be used by a Selector to describe a series of regions on the screen that represent "fields." On the other hand, the SelectField represents the main building block for the EntryField, lists of which form the main components of an EntryScreen. This in turn allows an EntryScreen to be derived from a Selector (actually an abstract object called an AbstractSelector), since any method that works strictly with the fields and methods of a SelectField will work equally well with an EntryField.

Although not an "abstract object" in the sense used elsewhere in this manual, a SelectField is indeed abstract in terms of the way that it describes a field. It defines the location and dimensions of a field, and the colors used to display it in different situations, but it contains no data concerning the contents of the field. It defines the location, width, and colors of a prompt that is to be associated with the field, but it keeps no record of the prompt itself. The reasons for thus limiting the amount of data stored in the SelectField will probably not become entirely clear to you until you've read further in this chapter. You can't fully understand the function served by a SelectField until you've read about EntryFields and Selectors. We can tell you, however, that the primary reason has to do with the fact that a Selector leaves more work to the programmer than an EntryScreen does. The programmer must provide a user-written routine to return the prompt for a particular field and a pre-formatted string representing its value, as well as an "action" routine that performs all editing tasks.

What follows is a discussion of all type and constant declarations pertaining primarily or exclusively to SelectFields.

Constants

```
BadSFieldOptions : LongInt = sfProtected+sfHidden+sfInvisible;
```

Internal SelectField options that should not be set directly.

```
DefSFieldOptions : LongInt = sfMapCtrls;
```

Default options for a SelectField.

```
sfProtected        = $00000001; {field is protected}
sfHidden           = $00000002; {field is hidden}
sfMapCtrls         = $00000004; {map control characters?}
sfIncDec           = $00000008; {field can be incremented/decremented}
sfInvisible        = $00000010; {if set, field is invisible}
```

These are the various options available for a SelectField. sfProtected indicates that the user may not move the cursor onto the field. sfHidden indicates that the field is not to be displayed, and the cursor may not be moved to it. Hiding is accomplished by drawing the field on the screen in a video attribute that has the foreground color set equal to the background color, with the background color being the same as the background color for the window in which the field is displayed. sfMapCtrls indicates that, when the field is displayed, control characters should be mapped to uppercase alphabetics (^A to 'A', for example) and displayed in a different video attribute. sfIncDec indicates that the field may be incremented by the ccIncChoice command or decremented by the ccDecChoice command. sfInvisible is similar to sfHidden, except that the field is never drawn at all. This option is generally used only for EntryFields. It allows multiple fields to share the same location on the screen; the programmer is responsible for deciding which one is visible at any given time.

Types

```
SelectFieldPtr = ^SelectField;
SelectField =
  object(AbstractField)
    sfID              : Word;    {field ID--assigned automatically}
```

```
    sfNextID          : Word;    {.$FFFF, or ID of next field}
    sfPrevID          : Word;    {.$FFFF, or ID of previous field}
    sfPRow, sfPCol    : Word;    {for prompt}
    sfPWidth          : Byte;
    sfFRow, sfFCol    : Word;    {for field}
    sfFWidth          : Byte;    {width of the field}
    sfFHeight         : Byte;    {height of the field}
    sfFCPos           : Byte;    {current column position within the field}
    sfFRPos           : Byte;    {current row position within the field}
    sfPromptColor     : Byte;    {for unselected prompt}
    sfPromptMono      : Byte;
    sfSelPromptColor  : Byte;    {for selected prompt}
    sfSelPromptMono   : Byte;
    sfProPromptColor  : Byte;    {for protected prompt}
    sfProPromptMono   : Byte;
    sfFieldColor      : Byte;    {for unselected field}
    sfFieldMono       : Byte;
    sfSelFieldColor   : Byte;    {for selected field}
    sfSelFieldMono    : Byte;
    sfProFieldColor   : Byte;    {for protected field}
    sfProFieldMono    : Byte;
    sfCtrlColor       : Byte;    {for control characters}
    sfCtrlMono        : Byte;
    sfOptions         : LongInt; {primary option settings}
    sfFlags           : LongInt; {secondary option settings}
    sfHelpIndex       : Word;    {index to pass to help routine}
    sfFieldName       : StringPtr; {for internal use}

    constructor Init(FieldID : Word; PromptRow, PromptCol : Word;
                     PromptWidth : Byte; FieldRow, FieldCol : Word;
                     fWidth, fHeight : Byte; HlpNdx : Word;
                     Options, IFlags : LongInt; var Colors : ColorSet);
    destructor Done; virtual;
    procedure sfOptionsOn(OptionFlags : LongInt);
      {-Activate multiple options}
    procedure sfOptionsOff(OptionFlags : LongInt);
      {-Deactivate multiple options}
    function sfOptionsAreOn(OptionFlags : LongInt) : Boolean;
      {-Return true if all specified options are on}
    procedure SetPromptAttr(Color, Mono : Byte);
      {-Set attributes for prompts}
    procedure SetSelectedPromptAttr(Color, Mono : Byte);
      {-Set attributes for selected prompts}
    procedure SetProtectedPromptAttr(Color, Mono : Byte);
      {-Set attributes for protected prompts}
    procedure SetFieldAttr(Color, Mono : Byte);
      {-Set attributes for unselected fields}
    procedure SetSelectedFieldAttr(Color, Mono : Byte);
      {-Set attributes for selected fields}
    procedure SetProtectedFieldAttr(Color, Mono : Byte);
      {-Set attributes for protected fields}
    procedure SetCtrlAttr(Color, Mono : Byte);
      {-Set attributes for control characters}
    constructor Load(var S : IdStream);
      {-Load a select field from a stream}
    procedure Store(var S : IdStream);
      {-Store a select field in a stream}
  end;
```

sfID is the ID number for the field, which is always assigned automatically by the Selector or EntryScreen that calls the field's constructor. The first field is always assigned 0, the next one 1, and so on. sfNextID and sfPrevID are normally set to $FFFF. If not, they contain the ID for the fields to move to when the ccTab/ccNextField or ccBackTab/ccPrevField commands are issued.

sfPRow and sfPCol are the relative coordinates for the field's prompt, if any. sfPWidth is its width. sfFRow and sfFCol are the relative coordinates for the field itself. sfFWidth is its width, and sfFHeight is its height. sfFCPos and sfFRPos are used internally to store the current location of the cursor within the field.

The next 14 fields hold the video attributes for the field itself, the associated prompt, and any control characters that appear within the field. sfOptions holds the primary option settings for the field, and sfFlags holds the secondary option settings. (sfFlags is currently used only by field types derived from SelectField.) sfHelpIndex is the help index for the field. sfFieldName is reserved for internal use (by MAKESCRN).

Other than Load, Init is the only constructor for a SelectField. Its job is mainly to make copies of values passed to it as parameters. It never allocates any memory. The Done constructor therefore has nothing to do unless Load has allocated a string pointed to by sfFieldName, something that happens only in the special entry screens created by MAKESCRN.

sfOptionsOn and sfOptionsOff activate or deactivate one or more field options. sfOptionsAreOn can be used to determine whether or not a particular option is in effect, or a group of related options are all on. It cannot be used to determine if one of several options is on.

SetPromptAttr sets the attributes to be used to display the field's prompt when the field is not selected and not protected. SetSelectedPromptAttr sets the attributes to be used to display the field's prompt when the field is selected. SetProtectedPromptAttr sets the attributes to be used to display the field's prompt when the field is protected.

SetFieldAttr sets the attributes to be used to display the field when it is not selected and not protected. SetSelectedFieldAttr sets the attributes to be used to display the field when it is selected. SetProtectedFieldAttr sets the attributes to be used to display the field when it is protected.

SetCtrlAttr sets the attributes to be used to display control characters within the field if the sfMapCtrls option is in effect.

Load and Store are needed only when a Selector or EntryScreen is being stored in or retrieved from a stream. You will never need to call them directly.

3. The EntryField

Unlike the SelectField, the EntryField is a true "abstract object." It exists to define the common denominators among all the specific field types–StringField, LongIntField, ChoiceField, MultiLineField–and thereby allow an EntryScreen to treat all fields derived from it as interchangeable.

This section is divided into five parts. The first part defines the term "driver routine," used frequently throughout the chapter, and explains some of the reasons for using driver routines in preference to virtual methods. The second part defines another group of frequently used terms: "regular field," "numeric field," and "simple field." The third and fourth parts describe the various options that affect the behavior of EntryFields. And the fifth describes the data fields and (documented) methods that distinguish an EntryField from its parent, the SelectField.

Methods vs. Driver Routines

One important term that arises frequently in our discussions of EntryFields is "driver routine." In order to explain what it means, we need to give you some background information.

There are four common operations performed on an EntryField that must or can be implemented differently depending on the type of field and other factors: drawing the field, editing the field, converting the field's value from its native type to a string and back again, and validating the field's contents during editing. Strings, Integers, Reals, Dates, Booleans–all require slightly different handling.

For each of these operations, EntryField declares both a method (Draw, Edit, Convert, Validate) and a procedure pointer (efDraw, efEditor, efConvert, efValidate). Ordinarily, the method simply makes an indirect call via the corresponding procedure pointer–for example, the default Draw method is essentially a one-line procedure consisting of a call to efDraw. In the case of certain derived types, one or more of these methods is overridden. DateField, for example, overrides the Convert and Validate methods, and MultiLineField overrides Draw, Edit, and Convert. In most cases, however, the method is not overridden, and the procedure pointer is set to point to a (somewhat) generic procedure called a "driver routine."

Using driver routines rather than overriding virtual methods would seem to go against the spirit of Object Oriented Programming, and it does. But there are some very good reasons for doing so. Consider, for example, the case of the field drawing routines. There are three different draw routines needed for the various field types: one for simple fields (SimpleDrawString), one for the MultiLineField (MultiLineField.Draw), and a third that can handle all the other field types (DrawString). An OOP purist would probably say that either SimpleDrawString or DrawString should be implemented as a method (EntryField.Draw), and that that method should be overridden as necessary. But doing so would almost ensure that code was linked in, even if it was never needed in the application.

The only way to prevent that from happening (while adhering strictly to the OOP philosophy) would be to define several abstract field objects derived from EntryField, with different Edit and Draw methods for each. We'll call this Plan F. The way we've implemented things, there is one LongIntField, derived from EntryField, that defines three constructors and uses three sets of driver routines (most of which are used for a variety of other field types as well). Under Plan F, the relevant portion of the object hierarchy would look something like this:

```
EntryField
  RegularField
    RegularLongIntField
  SimpleField
    SimpleLongIntField
  NumericField
    NumericLongIntField
```

rather than this:

```
EntryField
  LongIntField
```

The number of field objects would almost triple, code and data size would increase due to some lost opportunities to reuse code, and we'd have to distribute the diagram of the object hierarchy in the form of a wall chart.

There are other, similar reasons for using driver routines, but we won't enumerate them here. If you understood the last few paragraphs on first reading you can probably think of most of the other reasons yourself, since they all pertain to the issue of inheritance.

Regular, Numeric, and Simple Fields

OPFIELD distinguishes three classes of EntryFields--regular, numeric, and simple--based primarily on the field editor used for the field. At the lowest level, the classification depends on the constructor that is called to initialize the EntryField. Init instantiates a normal field; InitNum instantiates a numeric field; and InitSim instantiates a simple field.

A *regular field* generally uses one of two field editors, StringEditor or CharEditor (see the OPFEDIT section), both of which provide complete support for picture masks.

A *numeric field* is a field of a numeric type (real, BCD, integer, etc.) that is set up to use NumberEditor (see the OPFEDIT section), a calculator-style editor that allows numbers to be entered from right-to-left.

A *simple field* uses one of two field editors, SimpleStringEditor or SimpleCharEditor, neither of which supports true picture masks. These editors pay attention only to the first character in the picture mask, and use it only to determine which characters are valid for the field and whether or not characters entered by the user should be converted to upper or lower case.

In general, simple fields are best for line editing, while regular and numeric fields are best for data entry screens. It is possible, however, and in some cases highly desirable, to break the general rule. Simple fields pull significantly less code into an application than do regular fields, so they are preferable in applications that don't require picture masks. If an application does require picture masks for some fields, it is best to avoid using simple fields elsewhere if you can, since mixing the two will cause two field editors to be linked into the application where one would probably do.

This subject will be touched on again in the documentation for OPEDIT and OPENTRY.

Primary Field Options

As you know, most objects in **Object Professional** set aside 1-2 bytes of space to store option settings that affect the object's behavior. A few reserve 4 bytes (a LongInt) for that purpose. An EntryField reserves 8 bytes--two LongInts. The most commonly used option settings, stored in the first of the two LongInts, are referred to as primary field options, and have names that begin with a prefix of 'ef' (EntryField options). The less commonly used ones, stored in the

second LongInt, are referred to as secondary options and have a prefix of 'sef' (Secondary EntryField options). The primary options are discussed below, generally in groups of 4-5:

```
efProtected       = $00000001; {if set, field contents cannot be modified}
efHidden          = $00000002; {field is hidden}
efMapCtrls        = $00000004; {map control characters}
efMultChoice      = $00000008; {a multiple-choice field}
efInvisible       = $00000010; {if set, field is invisible}
```

These options correspond to the five SelectField options discussed earlier in the chapter. efProtected indicates that the user may not move the cursor onto the field. efHidden indicates that the field is not to be displayed, and the cursor may not be moved to it. Hiding is accomplished by drawing the field on the screen in a video attribute that has the foreground color set equal to the background color, with the background color being the same as the background color for the window in which the field is displayed. efMapCtrls indicates that, when the field is displayed, control characters should be mapped to uppercase alphabetics (^A to 'A', for example) and displayed in a different video attribute. efMultChoice is an internal flag indicating that the field is a ChoiceField; it should not be modified. efInvisible is similar to efHidden, except that the field is never drawn at all. This option allows multiple EntryFields to share the same location on the screen; the programmer is responsible for deciding which one is visible at any given time.

```
efAutoAdvCharBegin = $00000020; {dummy--for completeness}
efAutoAdvCharEnd   = $00000040; {advance to next field--character entered}
efAutoAdvCurBegin  = $00000080; {advance to prev field--cursor key pressed}
efAutoAdvCurEnd    = $00000100; {advance to next field--cursor key pressed}
efAutoAdvanceChar  = efAutoAdvCharBegin+efAutoAdvCharEnd;
efAutoAdvanceCursor= efAutoAdvCurBegin+efAutoAdvCurEnd;
efAutoAdvanceBegin = efAutoAdvCharBegin+efAutoAdvCurBegin;
efAutoAdvanceEnd   = efAutoAdvCharEnd+efAutoAdvCurEnd;
efAutoAdvance      = efAutoAdvanceBegin+efAutoAdvanceEnd;
```

These option settings all pertain to a feature referred to generically as "auto-advance." efAutoAdvCharEnd determines what happens during editing if the cursor is at the end of a field and the user enters another character: if the option is on, the cursor will be "auto-advanced" to the next field; if it is off, the cursor will remain where it is. efAutoAdvCurEnd determines what happens if the user is at the end of the field and issues a command that would normally move the cursor to the right: if the option is on, the cursor is advanced to the next field; if it is off, the cursor will remain where it is. Similarly, efAutoAdvCurBegin determines what happens if the user is at the start of the field and issues a command that would normally move the cursor to the left: if the option is on, the cursor is moved to the previous field; if it is off, the cursor will remain where it is. The efAutoAdvCharBegin option is currently unused, but may be needed in the future.

The constants efAutoAdvanceChar..efAutoAdvance provide a convenient means of referring to two or more of the first four options as a logical group. It is sometimes convenient, for example, to enable cursor-based auto-advance with 'efOptionsOn(efAutoAdvanceCursor)', and disable character-based auto-advance with 'efOptionsOff(efAutoAdvanceChar)', or to disable all forms of auto-advance with 'efOptionsOff(efAutoAdvance)'.

```
efInsertPushes    = $00000200; {inserting can push a character off end}
efRightJustify    = $00000400; {field is right-justified}
efPasswordMode    = $00000800; {contents of field are suppressed}
efCursorToEnd     = $00001000; {put cursor at end of string initially}
efTrimBlanks      = $00002000; {trim leading/trailing blanks when finished}
```

efInsertPushes is meaningful only for normal fields that do not use CharEditor (i.e., all normal fields except Char, Boolean, and YesNo). It indicates that, if the field is full, inserting a character can "push" the last character off the end of the field. efRightJustify indicates that the field is to be displayed right-justified. This option affects only normal fields; numeric fields are, by definition, right justified. efPasswordMode indicates that all meaningful characters in the field are to be suppressed, and represented with another character, called the PasswordChar. efCursorToEnd indicates that, by default, the cursor should be positioned

just beyond the last meaningful character in the field on entry to the field editor. efTrimBlanks indicates that any blank spaces at the end of a field should be stripped. This option has no effect if the picture mask for the field contains any literal or semi-literal mask characters.

```
efClearFirstChar   = $00004000; {clear string if first character is ASCII}
efForceOvertype    = $00008000; {if ForceMode set, selects insert or overtype}
efForceMode        = $00010000; {force insert or overtype, else use default}
efClickExit        = $00020000; {produce ccClickExit on double click}
```

efClearFirstChar indicates that the existing contents of the field should be cleared if the first key pressed during editing is a character. efForceMode indicates that either insert or overtype mode should be forced on (and not changed) when the field is being edited. If efForceMode is in effect, then efForceOvertype indicates whether the mode forced on is overtype or insert. efClickExit is meaningful only if the field is part of an EntryScreen. If the options is on, and the mouse is clicked on the current field (with the button assigned to ccMouseSel), the EntryScreen will exit and return ccClickExit as the last command.

```
efBeepOnError      = $00040000; {beep when illegal char entered?}
efAutoNumLock      = $00080000; {auto-activate NumLock--numbers only}
efParensForMinus   = $00100000; {show numbers < 0 inside ()--numbers only}
efExitAtEdges      = $00200000; {allow exit at edges--pick editor only}
efRequired         = $00400000; {if set, field cannot be empty}
```

efBeepOnError indicates whether or not the field editor should beep (call the RingBell routine in OPCMD) when the user enters an invalid command. If this option is on, and efInsertPushes and efAutoAdvCharEnd are both off, then the field editor will also emit a beep if the user tries to enter a character when the field is already full. efAutoNumLock allows you to specify that NumLock should temporarily be activated when the field is edited. This option is forced off for string fields and other non-numeric types. efRequired is meaningful only for fields in an EntryScreen. It indicates that the field may not be left blank.

efParensForMinus indicates that negative numbers should be displayed within parentheses--like so '(123.45)', rather than '-123.45'. This option applies only to numeric fields (those that use NumberEditor), and it requires that the picture mask have a blank space at the end, to be used for the closing ')' if necessary: for example, '######.## '.

```
efHouseCursorAtEnd = $01000000; {extra char of width to hold cursor}
```

This option is meaningful only for fields that use SimpleStringEditor (all simple fields except Char, Boolean, and YesNo). It indicates that an extra space should be reserved just beyond the end of the field to house the cursor when the field is full.

```
efForceUpper       = $02000000; {force chars to upper case?}
efForceLower       = $04000000; {force chars to lower case?}
```

These options are meaningful only for simple fields. efForceUpper indicates that all characters entered should be forced to upper case. efForceLower indicates that all characters entered should be forced to lower case. For non-simple fields, these effects can be accomplished only by means of the picture mask. For a simple field, these two options can override the case change characteristics of the field's picture mask.

```
efAllowEscape      = $08000000; {allow escape?}
```

This option is meaningful only to fields that use the SimpleCharEditor (simple Char, simple Boolean, and simple YesNo) and to PickFields (see OPENTRY). The flag indicates whether or not <Esc> (ccQuit) is an acceptable response. The flag is set by default, but you may wish to clear it, for example, when LineEditor.YesOrNo needs to force an answer of 'Y' or 'N'.

```
efShowReadChar     = $10000000; {display the character pressed?}
efHideCursor       = $20000000; {hide cursor?}
efDefaultAccepted  = $40000000; {allow user to accept default?}
efMapExtended      = $80000000; {map extended keys?}
```

These options are meaningful only to fields that use the SimpleCharEditor (simple Char, Boolean, and YesNo). Their default settings should not be changed when working with EntryScreens. They are intended to be used only in conjunction with a LineEditor.

Secondary Field Options and Internal Flags

The second group of field options actually consists of two kinds of bit flags: secondary field options ('sef' prefix) and internal flags ('if' prefix). Both kinds of flags are discussed below:

```
ifBoolean        = $00000001; {flags a boolean or yes/no field}
ifNumeric        = $00000002; {field uses numeric editor}
ifSimple         = $00000004; {flags a "simple" field}
ifNested         = $00000008; {field is assoc with a nested entry screen}
ifMultiLine      = $00000010; {flags a multi-line field}
ifModified       = $00000020; {marks fields modified while editing}
ifNotNext        = $00000040; {disallow ccNextField}
ifNotPrev        = $00000080; {disallow ccPrevField}
ifNoMouseExit    = $00000100; {ccMouseSel can't exit (multi-line only)}
ifSuppressFirst  = $00000200; {don't display first time--simple char only}
ifNoLiterals     = $00000400; {picture mask has no literal characters}
ifEditing        = $00000800; {set while field is being edited}
ifHexadecimal    = $00001000; {if set, field's value is shown in hex}
ifSemiHidden     = $00002000; {if set, field is semi-hidden}
ifRealVar        = $00004000; {if set, field is of a real/BCD/8087 real}
```

These options are all internal flags that should not be modified unless absolutely necessary. The only one that should be of general interest is ifModified, a flag set by a field editor on exit if the field was modified by the user.

```
sefPadCurrentOnly  = $08000000; {use pad character only when editing}
sefSuppressZero    = $10000000; {suppress display of 0's for integer types}
sefNoFieldMovement = $20000000; {don't allow field movement commands to exit}
```

sefPadCurrentOnly indicates that the special pad character for a field ('_' for example) should be used only while the field is being edited; if the field is not being edited, ' ' will be used instead. sefSuppressZero is meaningful only for a field of an integer type (Byte, ShortInt, Integer, Word, LongInt). It indicates that the field should blanked out (semi-hidden) when the field is not being edited. That is, if a Word field's value is 0, it is treated the same way as a Real field whose value is BadReal. sefNoFieldMovement is an option normally used only by a LineEditor. It tells a field editor not to exit when a field movement command such as ccDown is issued. The only other case where it is useful is that of an EntryScreen with a single field.

```
sefWindowRelative  = $40000000; {coords treated relative to current window?}
sefInsertByDefault = $80000000; {default to insert mode?}
```

These last two options are meaningful only to a LineEditor. They are declared in OPFIELD only to reserve space for the two flags.

What follows is a discussion of all other type and constant declarations pertaining primarily or exclusively to EntryFields.

Constants

```
BadEFieldOptions   : LongInt = efProtected+efMultChoice;
```

Primary EntryField options that should not be changed directly.

```
BadSEFieldOptions  : LongInt = ifBoolean+ifNumeric+ifSimple+ifNested+
                               ifMultiLine+ifModified+ifNotNext+ifNotPrev+
                               ifNoMouseExit+ifSuppressFirst+ifNoLiterals+
                               ifEditing+ifHexadecimal+ifSemiHidden+ifRealVar;
```

Secondary EntryField options that should not be changed directly.

```
DefEFieldOptions : LongInt = efInsertPushes+efAutoAdvance+efTrimBlanks+
                             efMapCtrls+efAllowEscape+efDefaultAccepted+
                             efShowReadChar;
```

Default primary EntryField options. Note that certain derived types may force some of these options off or activate additional ones. An ArrayField, for example, always forces efTrimBlanks off.

```
DefSEFieldOptions : LongInt = 0;
```

Default secondary EntryField options. Note that certain derived types may force some of these options off or activate additional ones.

```
DefPadChar : Char = ' ';
```

Default character used to pad the end of a partially empty field.

```
DefPasswordChar : Char = ^G;
```

Default character used to represent the text in a field while in password mode.

Types

```
ConversionProc = procedure (EFP : EntryFieldPtr; PostEdit : Boolean);
```

A conversion routine written in the form of a driver rather than a method. See section 5, "Field Conversion Routines," for a discussion of conversion routines.

```
DrawProc = procedure (var Field; St : string; Row, Col : Word;
                      FA, CA, POffset : Byte; PasswordChar : Char;
                      Flags : PictureFlags);
```

A draw routine written in the form of a driver rather than a method. See section 7, "Field Drawing Routines," for a discussion of draw routines.

```
EditProc = procedure (var Field; Row, Col : Word; FA, CA : Byte;
                      PasswordChar : Char; PosCode : Byte;
                      ReadOnly : Boolean; var CC : Word; var ChWord : Word;
                      var InsertMode : Boolean; ErrorRoutine : ErrorProc;
                      UnitCode : Byte; var CP : CommandProcessor);
```

A field editor written in the form of a driver rather than a method. See the section on OPFEDIT for a discussion of field editors.

```
RangeType =
  record case Byte of                    {size}
    00 : (rtChar : Char);                 {01}
    01 : (rtByte : Byte);                 {01}
    02 : (rtSht  : ShortInt);             {01}
    03 : (rtInt  : Integer);              {02}
    04 : (rtWord : Word);                 {02}
    05 : (rt3    : array[1..3] of Byte);  {03} {for type-casting}
    06 : (rtLong : LongInt);              {04}
    07 : (rtSgl  : Single);               {04}
    08 : (rtPtr  : Pointer);              {04}
    09 : (rt5    : array[1..5] of Byte);  {05} {for type-casting}
    10 : (rtReal : Real);                 {06}
    11 : (rt7    : array[1..7] of Byte);  {07} {for type-casting}
    12 : (rtDbl  : Double);               {08}
    13 : (rtComp : Comp);                 {08}
    14 : (rt9    : array[1..9] of Byte);  {09} {for type-casting}
    15 : (rtExt  : Extended);             {10}
    {$IFDEF UseBcd}
    16 : (rtBcd  : BCD);                  {10}
    {$ENDIF}
    17 : (rt10   : array[1..10] of Byte); {10} {for type-casting}
    {$IFDEF UseDates}
    18 : (rtDate : Date);                 {2 or 4}
    19 : (rtTime : Time);                 {04}
    {$ENDIF}
  end;
```

An EntryField contains two records of this type, one to store the lower limit for a field's value, and another to store the upper limit. The variant record is set up in such a way as to allow any data type of no more than 10 bytes to be stored in it. To treat the efRangeLo field of an EntryField as a Real, for example, you would refer to 'efRangeLo.rtReal'. ('rt' stands for Range Type.)

```
ValidationFunc = function (EFP : EntryFieldPtr; var ErrCode : Word;
                           var ErrorSt : StringPtr) : Boolean;
```

A validation routine written in the form of a driver rather than a method. See section 6 of the OPFIELD documentation, "Field Validation Routines," for a discussion of validation routines.

```
EntryFieldPtr = ^EntryField;
EntryField =
  object(SelectField)
    efPrompt   : StringPtr;      {points to edit prompt}
    efPicture  : StringPtr;      {points to edit mask}
    efEditSt   : StringPtr;      {points to string to be edited}
    efValidate : ValidationFunc; {pointer to validation routine, or nil}
    efConvert  : ConversionProc; {points to conversion routine}
    efDraw     : DrawProc;       {routine to draw the field}
    efEditor   : EditProc;       {editor for this field}
    efMaxLen   : Byte;           {maximum length of string}
    efPadChar  : Char;           {char used to pad end of string}
    efDPlaces  : Byte;           {max decimal places, if no '.' in Picture}
    efRangeLo  : RangeType;      {valid range for the field}
    efRangeHi  : RangeType;
    efDataSize : Word;           {size of data type}
    efVarPtr   : Pointer;        {points to the actual variable}

    constructor Init(ID : Word; var Prompt : string; pRow, pCol : Word;
                     var Picture : string; fRow, fCol : Word;
                     fWidth, fHeight : Byte; HlpNdx : Word;
                     var RangeLo, RangeHi : RangeType; DataSize : Word;
                     DecimalPlaces : Byte; VF : ValidationFunc;
                     CP : ConversionProc; DP : DrawProc; EP : EditProc;
                     var EditVar; PadChar : Char; Options, IFlags : LongInt;
                     var Colors : ColorSet);
      {-Initialize an entry field}
    constructor InitNPP(ID : Word; var Prompt : string; pRow, pCol : Word;
                        var Picture : string; fRow, fCol : Word;
                        fWidth, fHeight : Byte; HlpNdx : Word;
                        var RangeLo, RangeHi : RangeType; DataSize : Word;
                        DecimalPlaces : Byte; var EditVar; PadChar : Char;
                        Options, IFlags : LongInt; var Colors : ColorSet);
      {-Initialize an entry field (No Procedure Pointers)}
    destructor Done; virtual;
      {-Deallocate dynamic data}

    procedure efOptionsOn(OptionFlags : LongInt);
      {-Activate multiple options}
    procedure efOptionsOff(OptionFlags : LongInt);
      {-Deactivate multiple options}
    function efOptionsAreOn(OptionFlags : LongInt) : Boolean;
      {-Return true if all specified options are on}
    procedure sefOptionsOn(OptionFlags : LongInt);
      {-Activate multiple secondary options}
    procedure sefOptionsOff(OptionFlags : LongInt);
      {-Deactivate multiple secondary options}
    function sefOptionsAreOn(OptionFlags : LongInt) : Boolean;
      {-Return true if all specified secondary options are on}
```

```
    procedure Convert(PostEdit : Boolean); virtual;
      {-Convert from native type to edit string or vice versa}
    procedure Draw(var St : string; Row, Col : Word; FA, CA, POffset : Byte;
                   PasswordChar : Char; var Flags : PictureFlags); virtual;
      {-Draw the field (St has string to draw)}
    procedure Edit(Row, Col : Word; FA, CA : Byte; PasswordChar : Char;
                   PosCode : Byte; ReadOnly : Boolean; var CC, ChWord : Word;
                   var InsertMode : Boolean; EP : ErrorProc;
                   UnitCode : Byte; var CP : CommandProcessor); virtual;
      {-Edit the field}
    function Validate(var Code : Word; var SP : StringPtr) : Boolean; virtual;
      {-Validate the field}
    constructor Load(var S : IdStream);
      {-Load an entry field from a stream}
    procedure Store(var S : IdStream);
      {-Store an entry field in a stream}
  end;
```

efPrompt points to the prompt for the field. It is never nil; if the field has no prompt, efPrompt prompt points to an empty string (a single byte set to 0, representing the length byte of the string). efPicture points to the picture mask for the field. Note that even "simple" fields have a full-size picture mask; all the characters in the mask are the same as the first character. efEditSt points to a formatted string representing the field's value. This string is not initialized until the field's conversion routine has been callled.

The next four fields are pointers to driver routines. efValidate points to the validation routine, efConvert to the conversion routine, efDraw to the draw routine, and efEditor to the field editor. The procedures pointed to should never be called directly. The corresponding method (Validate, Convert, Draw, and Edit respectively) should be called instead; it will then route the call to the driver routine if the method has not been overridden.

efMaxLen is the maximum length of the string to be edited. In most cases, it is equal to the width of the field (sfFWidth), but if the field can be scrolled horizontally efMaxLen will be greater than sfFWidth. Length(efEditSt^) and Length(efPicture^) will always be equal to efMaxLen. efPadChar is the character used to pad the end of the field, if necessary, when it is displayed.

If the field is of a real data type (Real, BCD, Extended, etc.), and the picture mask does not contain a fixed decimal point, then efDPlaces is the maximum number of decimal places to be displayed. efRangeLo and efRangeHi define the valid range for the field. efVarPtr points to the variable associated with the field, and efDataSize is its size.

The Init and InitNPP constructors are essentially identical. Their main tasks are to make copies of the values passed to them and as parameters and to allocate space for the prompt (efPrompt^), picture mask (efPicture^), and the string used during editing (efEditSt^). The main difference between them is that InitNPP does not accept any procedure pointers as parameters (NPP = No Procedure Pointers). It is called by the constructors for derived field types that override one or more of the following methods: Edit, Draw, Convert, or Validate. Init is called by the constructors for field types that rely strictly on driver routines. Done simply deallocates the memory previously allocated by a constructor.

efOptionsOn and efOptionsOff activate or deactivate one or more primary field options. efOptionsAreOn can be used to determine whether or not a particular primary option is in effect, or a group of related primary options are all on. It cannot be used to determine if one of several options is on. sefOptionsOn, sefOptionsOff, sefOptionsAreOn serve the same three functions, but apply to secondary options rather than primary options.

Convert is always called just before a field is drawn, and both before and after a field is edited. Its job is to convert the value associated with the field (efVarPtr^) to a formatted string (efEditSt^) or vice versa, depending on the value of PostEdit. (See section 5, "Field Conversion Routines.")

Draw displays the string passed to it at the specified location. (See section 7, "Field Drawing Routines.")

Edit is called when the field needs to be edited. (See the section on OPFEDIT for a discussion of field editors.)

Validate is called by the field editor to determine whether or not a string entered by the user (efEditSt^) is an acceptable response. (See section 6, "Field Validation Routines.")

Load and Store are needed only when an EntryScreen is being stored in or retrieved from a stream. You will never need to call them directly.

4. Derived Field Types

This section covers the numerous field types that are derived from the generic EntryField. The entry for each field type consists of three parts. The first part is the type declaration for the field. It generally consists of an empty block, except in cases where an extra data field is added or one of EntryField's virtual methods is overridden. The list of methods overridden is never complete, however, because the third part of each entry is a brief table showing the constructor(s) for the field, and the type declaration shows only those overridden methods that are not mentioned in the table.

Each table is of the following form:

```
ConstructorName
  Draw:     name of driver or '--overrides method--'
  Edit:     name of driver or '--overrides method--'
  Validate: name of driver, '--none--', or '--overrides method--'
  Convert:  name of driver or '--overrides method--'
  Register: name of stream registration routine
```

These tables are intended to give the clearest possible explanation of what the differences are among the various field types.

The second part of each entry is just a brief description of the field type.

ArrayField

```
ArrayFieldPtr = ^ArrayField;
ArrayField =
  object(EntryField)
  end;
```

This field type works with arrays of char, which are commonly used in database applications instead of string fields for storing ordinary text. In such a field, the array is always filled to capacity; unused slots at the end of the array are padded with blanks (' ').

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: --none--
  Convert:  --overrides method--
  Register: ArrayFieldStream
```

BcdField

```
{$IFDEF UseBcd}
BcdFieldPtr = ^BcdField;
BcdField =
  object(EntryField)
  end;
```

This field type is used for BCD reals created with the OPBCD unit. It is implemented only if the conditional compilation directive UseBcd is enabled.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateBcd
  Convert:  BcdConversion
  Register: BcdFieldStream
```

```
InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateBcd
  Convert:  BcdConversion
  Register: BcdFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateBcd
  Convert:  SimpleBcdConversion
  Register: BcdFieldSimStream
```

BooleanField

```
BooleanFieldPtr = ^BooleanField;
BooleanField =
   object(EntryField)
      procedure efIncrement; virtual;
   end;
```

This field type is used for Boolean variables, which are normally represented on screen as 'T' (True) or 'F' (False). During editing, a BooleanField is essentially treated as a special instance of a CharField. Note that BooleanField overrides the efIncrement method to allow a mouse click to toggle the value of the associated variable.

```
Init
  Draw:     DrawString
  Edit:     CharEditor
  Validate: NullValidation
  Convert:  BooleanConversion
  Register: BooleanFieldStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleCharEditor
  Validate: NullValidation
  Convert:  SimpleBooleanConversion
  Register: BooleanFieldSimStream
```

ByteField

```
ByteFieldPtr = ^ByteField;
ByteField =
  object(EntryField)
  end;
```

This field type is used for variables of type Byte. Note that the upper and lower limits for such a field are stored as LongInts, allowing ByteField to use the same validation routines as a LongIntField.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateLong
  Convert:  ByteConversion
  Register: ByteFieldStream
```

```
InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateLong
  Convert:  ByteConversion
  Register: ByteFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateLong
  Convert:  SimpleByteConversion
  Register: ByteFieldSimStream
```

CharField

```
CharFieldPtr = ^CharField;
CharField =
 object(EntryField)
 end;
```

This field type is used for variables of type Char. Note in the table below that the choice of validation routines depends on whether or not the programmer has specified upper and lower limits for the field. If no limits were specified, NullValidation is used.

```
Init
  Draw:     DrawString
  Edit:     CharEditor
  Validate: ValidateChar or NullValidation
  Convert:  CharConversion
  Register: CharFieldStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleCharEditor
  Validate: SimpleValidateChar or NullValidation
  Convert:  SimpleCharConversion
  Register: CharFieldSimStream
```

ChoiceField

```
ChoiceFieldPtr = ^ChoiceField;
IncChoiceProc = procedure (var Value;        FieldID : Word;
                           Factor : Integer; var St : string);
ChoiceField =
  object(EntryField)
    cfIncChoice : IncChoiceProc;

    constructor Load(var S : IdStream);
    procedure Store(var S : IdStream);
    procedure efIncrement; virtual;
  end;
```

This field type is normally used for variables of enumerated types, but can also be used for numeric variables in cases where the value is to be incremented or decremented by the user rather than entered. Each ChoiceField stores a pointer to a user-written routine of type IncChoiceProc, which takes four parameters: an untyped var parameter corresponding to the variable to be incremented/decremented; a field ID, to allow a single IncChoiceProc to be used for multiple fields; Factor, a value in the range -1..1, which tells the routine whether to increment Value, decrement it, or leave it alone; and St, which must contain on exit a string representation of the field's value. The IncChoiceProc can be called by any one of three

methods: ChoiceField.Edit (field editor), ChoiceField.Convert (conversion routine), and ChoiceField.efIncrement (called when user clicks mouse on the field to increment its value by a factor of 1).

ChoiceField uses a special purpose field editor, implemented as a method, that treats all cursor movement commands (such as ccLeft) as exit commands. The only "editing" commands it responds to are ccDecChoice (decrement choice), ccIncChoice (increment choice), and ccRestore (restore default value and continue).

```
Init
  Draw:     DrawString
  Edit:     --overrides method--
  Validate: --none--
  Convert:  --overrides method--
  Register: ChoiceFieldStream
```

CompField

```
{$IFOPT N+}
CompFieldPtr = ^CompField;
CompField =
  object(EntryField)
  end;
```

This field type is used for variables of type Comp. It is implemented only if OPFIELD was compiled with {$N+}.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateExt
  Convert:  CompConversion
  Register: CompFieldStream

InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateExt
  Convert:  CompConversion
  Register: CompFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateExt
  Convert:  SimpleCompConversion
  Register: CompFieldSimStream
```

DateField

```
{$IFDEF UseDates}
DateFieldPtr = ^DateField;
DateField =
  object(EntryField)
  end;
```

This field type is used for Date variables created with the OPDATE unit. It is implemented only if the conditional compilation directive UseDates is enabled.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: --overrides method--
  Convert:  --overrides method--
  Register: DateFieldStream
```

DateStField

```
{$IFDEF UseDates}
DateStFieldPtr = ^DateStField;
DateStField =
  object(EntryField)
  end;
```

This field type is used for DateString variables created with the OPDATE unit. It is implemented only if the conditional compilation directive UseDates is enabled.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: --overrides method--
  Convert:  --overrides method--
  Register: DateStFieldStream
```

DoubleField

```
{$IFOPT N+}
DoubleFieldPtr = ^DoubleField;
DoubleField =
  object(EntryField)
  end;
```

This field type is used for variables of type Double. It is implemented only if OPFIELD was compiled with {$N+}.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateExt
  Convert:  DblConversion
  Register: DoubleFieldStream

InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateExt
  Convert:  DblConversion
  Register: DoubleFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateExt
  Convert:  SimpleDblConversion
  Register: DoubleFieldSimStream
```

ExtendedField

```
{$IFOPT N+}
ExtendedFieldPtr = ^ExtendedField;
ExtendedField =
  object(EntryField)
  end;
```

This field type is used for variables of type Extended. It is implemented only if OPFIELD was compiled with {$N+}.

```
    Init
      Draw:     DrawString
      Edit:     StringEditor
      Validate: ValidateExt
      Convert:  ExtConversion
      Register: ExtendedFieldStream

    InitNum
      Draw:     DrawString
      Edit:     NumberEditor
      Validate: ValidateExt
      Convert:  ExtConversion
      Register: ExtendedFieldNumStream

    InitSim
      Draw:     SimpleDrawString
      Edit:     SimpleStringEditor
      Validate: SimpleValidateExt
      Convert:  SimpleExtConversion
      Register: ExtendedFieldSimStream
```

IntegerField

```
IntegerFieldPtr = ^IntegerField;
IntegerField =
  object(EntryField)
  end;
```

This field type is used for variables of type Integer. Note that the upper and lower limits for such a field are stored as LongInts, allowing IntegerField to use the same validation routines as a LongIntField.

```
    Init
      Draw:     DrawString
      Edit:     StringEditor
      Validate: ValidateLong
      Convert:  IntConversion
      Register: IntegerFieldStream

    InitNum
      Draw:     DrawString
      Edit:     NumberEditor
      Validate: ValidateLong
      Convert:  IntConversion
      Register: IntegerFieldNumStream

    InitSim
      Draw:     SimpleDrawString
      Edit:     SimpleStringEditor
      Validate: SimpleValidateLong
      Convert:  SimpleIntConversion
      Register: IntegerFieldSimStream
```

LongIntField

```
LongIntFieldPtr = ^LongIntField;
LongIntField =
  object(EntryField)
  end;
```

This field type is used for variables of type LongInt.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateLong
  Convert:  LongConversion
  Register: LongIntFieldStream

InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateLong
  Convert:  LongConversion
  Register: LongIntFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateLong
  Convert:  SimpleLongConversion
  Register: LongIntFieldSimStream
```

MultiLineField

```
MultiLineFieldPtr = ^MultiLineField;
MultiLineField =
  object(EntryField)
  end;
```

This field type is used for editing arrays of strings. The goal in developing it was to provide a simplistic alternative to using OPMEMO to implement memo fields. In contrast to memo fields, which require the programmer to add a dummy nested field, and the user to issue a special exit command to invoke the Memo, a MultiLineField looks and acts much like any other built-in field type.

The Edit method implemented by MultiLineField allows the user to edit multiple lines of text without moving from one field to another. Lines can be inserted, deleted, and joined (if space allows), and the mouse can be used to move the cursor from one part of the field to another. Cursor movement commands such as ccLeft and ccRight always move the cursor from one part of the field to another (if at all) unless the cursor is at the edge of the field. (ccLeft and ccRight move the cursor to another field only if the cursor is the top left or bottom right corner of the field, respectively.) One noteworthy thing the editor cannot do is scroll the contents of the field: if the field is 5 rows high and 60 columns wide, the variable being edited is assumed to be an 'array[1..5] of string[60]', and is always displayed in its entirety.

```
Init
  Draw:     --overrides method--
  Edit:     --overrides method--
  Validate: --none--
  Convert:  --overrides method--
  Register: MultiLineFieldStream
```

NestedField

```
NestedFieldPtr = ^NestedField;
NestedField =
  object(EntryField)
  end;
```

This special field type is used to create "dummy" fields that mark links to nested entry screens (a second entry screen to be displayed on top of the first) or memo fields. A nested field is typically 1-2 characters wide, and the only thing the user can do when on such a field is execute a field movement command or an exit command. Generally, a nested field corresponds to a special exit command that will invoke the nested entry screen or memo field.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: --none--
  Convert:  --overrides method--
  Register: NestedFieldStream.
```

RealField

```
RealFieldPtr = ^RealField;
RealField =
  object(EntryField)
  end;
```

This field type is used for variables of type Real.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateReal
  Convert:  RealConversion
  Register: RealFieldStream

InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateReal
  Convert:  RealConversion
  Register: RealFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateReal
  Convert:  SimpleRealConversion
  Register: RealFieldSimStream
```

ShortIntField

```
ShortIntFieldPtr = ^ShortIntField;
ShortIntField =
  object(EntryField)
  end;
```

This field type is used for variables of type ShortInt. Note that the upper and lower limits for such a field are stored as LongInts, allowing ShortIntField to use the same validation routines as a LongIntField.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateLong
  Convert:  ShortConversion
  Register: ShortIntFieldStream
```

```
    InitNum
      Draw:     DrawString
      Edit:     NumberEditor
      Validate: ValidateLong
      Convert:  ShortConversion
      Register: ShortIntFieldNumStream

    InitSim
      Draw:     SimpleDrawString
      Edit:     SimpleStringEditor
      Validate: SimpleValidateLong
      Convert:  SimpleShortConversion
      Register: ShortIntFieldSimStream
```

SingleField

```
{$IFOPT N+}
SingleFieldPtr = ^SingleField;
SingleField =
  object(EntryField)
  end;
```

This field type is used for variables of type Single. It is implemented only if OPFIELD was compiled with {$N+}.

```
    Init
      Draw:     DrawString
      Edit:     StringEditor
      Validate: ValidateExt
      Convert:  SglConversion
      Register: SingleFieldStream

    InitNum
      Draw:     DrawString
      Edit:     NumberEditor
      Validate: ValidateExt
      Convert:  SglConversion
      Register: SingleFieldNumStream

    InitSim
      Draw:     SimpleDrawString
      Edit:     SimpleStringEditor
      Validate: SimpleValidateExt
      Convert:  SimpleSglConversion
      Register: SingleFieldSimStream
```

StringField

```
StringFieldPtr = ^StringField;
StringField =
  object(EntryField)
  end;
```

This field type is used for variables of type String (any length).

```
    Init
      Draw:     DrawString
      Edit:     StringEditor
      Validate: NullValidation
      Convert:  StringConversion
      Register: StringFieldStream
```

```
InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: NullValidation
  Convert:  SimpleStringConversion
  Register: StringFieldSimStream
```

TimeField

```
{$IFDEF UseDates}
TimeFieldPtr = ^TimeField;
TimeField =
  object(EntryField)
  end;
```

This field type is used for Time variables created with the OPDATE unit. It is implemented only if the conditional compilation directive UseDates is enabled.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: --overrides method--
  Convert:  --overrides method--
  Register: TimeFieldStream
```

WordField

```
WordFieldPtr = ^WordField;
WordField =
  object(EntryField)
  end;
```

This field type is used for variables of type Word. Note that the upper and lower limits for such a field are stored as LongInts, allowing WordField to use the same validation routines as a LongIntField.

```
Init
  Draw:     DrawString
  Edit:     StringEditor
  Validate: ValidateLong
  Convert:  WordConversion
  Register: WordFieldStream

InitNum
  Draw:     DrawString
  Edit:     NumberEditor
  Validate: ValidateLong
  Convert:  WordConversion
  Register: WordFieldNumStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleStringEditor
  Validate: SimpleValidateLong
  Convert:  SimpleWordConversion
  Register: WordFieldSimStream
```

YesNoField

```
YesNoFieldPtr = ^YesNoField;
YesNoField =
  object(EntryField)
    procedure efIncrement; virtual;
  end;
```

This field type is used for Boolean variables that are to be represented on screen in terms of Yes/No ('Y' or 'N') rather than True/False ('T' or 'F'). During editing, a YesNoField is essentially treated as a special instance of a CharField. Note that YesNoField overrides the efIncrement method to allow a mouse click to toggle the value of the associated variable.

```
Init
  Draw:     DrawString
  Edit:     CharEditor
  Validate: NullValidation
  Convert:  YesNoConversion
  Register: YesNoFieldStream

InitSim
  Draw:     SimpleDrawString
  Edit:     SimpleCharEditor
  Validate: NullValidation
  Convert:  SimpleYesNoConversion
  Register: YesNoFieldSimStream
```

5. Field Conversion Routines

The end of this section contains a table showing all the standalone conversion routines used by the various derived field types discussed in the previous section. The real purpose of the section, however, is to explain what a conversion routine does and to give you some of the knowledge required to write new ones.

A conversion routine has the following form:

```
{$F+}
procedure MyConversionProc(EFP : EntryFieldPtr; PostEdit : Boolean);
begin
  with EFP^ do
    if PostEdit then
      {convert efEditSt^ to efVarPtr^}
    else
      {convert efVarPtr^ to efEditSt^} ;
end;
$F-}
```

The EFP parameter is a pointer to the field to be converted. The PostEdit parameter will be True if the user has just finished editing the field. It means that the contents of the string pointed to by efEditSt needs to be converted into the field's native type and then stored in the variable pointed to by efVarPtr. If PostEdit is False, the field is about to be displayed, and the variable needs to be converted from its native type into a formatted string, stored in efEditSt^.

To give you a better idea of how these things work, let's look at one of them. We'll use WordConversion as our example, since it demonstrates the basic ideas pretty well.

```
procedure WordConversion(EFP : EntryFieldPtr; PostEdit : Boolean);
  {-Conversion routine for words}
var
  S : string[80];
  L : LongInt;
begin
  with EFP^ do begin
    if PostEdit then begin
      StripPicture(efEditSt^, S);
      if not efStr2Long(S, L) then
        Word(efVarPtr^) := 0
      else
        Word(efVarPtr^) := Word(L);
    end
    else begin
      S := efLong2Str(Word(efVarPtr^), 4);
      MergePicture(S, efEditSt^);
    end;

    efSetSemiHidden((Word(efVarPtr^) = 0) and
                    LongFlagIsSet(sfFlags, sefSuppressZero));
  end;
end;
```

We'll start with the case where PostEdit is False (the second half of the routine). Since a Word is a numeric type, and the goal is to produce a formatted string, the first task is to convert the Word variable pointed to by efVarPtr into a string. For this we use efLong2Str, a static "helper" method declared by EntryField, that takes two parameters: a long integer value and a string width. This routine will call one of two routines in OPSTRING, HexL or Long2Str, based on

the picture mask in use. If the picture mask indicates that the field is a hexadecimal value, then the string width tells efLong2Str how many hex digits should appear in the string (2 for Byte/ShortInt, 4 for Word/Integer, 8 for LongInt). Once the value has been converted to a raw string (S), the MergePicture method inherited from AbstractField is called to merge its contents with that of the picture mask and store the result in efEditSt^.

If PostEdit is True, the process is almost precisely reversed. The first task is to strip out extraneous characters added by MergePicture (and retained during editing) by calling the StripPicture method inherited from AbstractField. The raw string it returns (S) will be of the same form as the one passed to MergePicture. The next step is to call efStr2Long, another helper method declared by EntryField, to convert the string (first parameter) into a value of type LongInt (second parameter). Like efLong2Str, it handles fields displayed in either hexadecimal or decimal form. If efStr2Long returns True, the conversion was successful and the value returned is stored in efVarPtr^; if it returns False (something that shouldn't happen), efVarPtr^ is set to 0 instead.

The last statement in the routine is a call to efSetSemiHidden, a method declared by EntryField and used only by conversion routines. It is normally used to flag fields that contain "illegal" values (BadReal, BadDate, BadTime, etc.) In the case of integer-type fields, however, it is used to flag a field that contains a value of zero--if the sefSuppressZero option is in effect for that field. The effect of the internal flag set by efSetSemiHidden is to suppress the display of the field's value when it is not being edited.

We can't pretend that our discussion of this one example tells you all you need to know in order to write your own conversion routines. There is another example in the documentation for OPENTRY, we might add, but it is very much like this one. Other data types require slightly different steps than those shown here, especially reals, and a conversion routine for a "simple" field is very different indeed, because it doesn't have a picture mask to deal with. Nevertheless, this is a fairly typical example, and it should give you a general idea of what needs to be done in such a routine. If you need to write one, our best advice is to study the numerous examples found in the source code for OPFIELD. Find the one that comes closest to what you think yours needs to do, make a copy of it, and then modify it to do exactly what you want.

Built-in Conversion Routines

The table below shows all the standalone conversion routines in OPFIELD. If no conversion routine is shown for a particular field type, then conversion for the field is performed by a method rather than a driver routine. The first column shows the generic name of the field type (add 'Field' to the end of the generic name to get the actual name). The next three columns show the conversion routines that are used in different situations. "Regular" is the one normally used for a field with a picture mask. "Numeric" is the one used if the field is being edited with the NumberEditor; in the cases where this is possible, it is always the same as the "regular" conversion routine. "Simple" is the one used if the field lacks a true picture mask. The names in parentheses indicate the corresponding constructor. For example, if a ByteField was initialized by the InitSim constructor, the corresponding conversion routine would be SimpleByteConversion.

Field Type	Regular (Init)	Numeric (InitNum)	Simple (InitSim)
BCD	BcdConversion	BcdConversion	SimpleBcdConversion
Boolean	BooleanConversion	---	SimpleBooleanConversion
Byte	ByteConversion	ByteConversion	SimpleByteConversion
Char	CharConversion	---	SimpleCharConversion
Comp	CompConversion	CompConversion	SimpleCompConversion
Double	DblConversion	DblConversion	SimpleDblConversion
Extended	ExtConversion	ExtConversion	SimpleExtConversion
Integer	IntConversion	IntConversion	SimpleIntConversion
LongInt	LongConversion	LongConversion	SimpleLongConversion
Real	RealConversion	RealConversion	SimpleRealConversion
Single	SglConversion	SglConversion	SimpleSglConversion
ShortInt	ShortConversion	ShortConversion	SimpleShortConversion
String	StringConversion	---	SimpleStringConversion
Word	WordConversion	WordConversion	SimpleWordConversion
YesNo	YesNoConversion	---	SimpleYesNoConversion

6. Field Validation Routines

Validation routines are called by field editors to verify that a string entered by the user represents an acceptable response. What does a validation routine do? In the case of numeric fields, a validation routine would need to do two things: verify that the string entered actually represents a number of the proper form, and verify that that number is within an acceptable range of values. Although OPFIELD provides all the validation routines most people will ever need, we should probably look at one closely to get a general idea of how they work, in case you ever need to write one of your own.

A validation routine has the following basic form:

```
{$F+}
function MyValidationFunc(EFP : EntryFieldPtr; var ErrCode : Word;
                          var ErrorSt : StringPtr) : Boolean;
begin
  with EFP^ do begin
    {try to convert efEditSt^ to native type}
    {if conversion fails then begin}
      {MyValidationFunc := False;}
      {set ErrCode and ErrorSt}
    {end}
    {else if value in acceptable range then}
      {MyValidationFunc := True;}
    {else begin}
      {MyValidationFunc := False;}
      {set ErrCode and ErrorSt}
    {end;}
  end;
end;
{$F-}
```

The EFP parameter is a pointer to the field to be validated, and EFP^.efEditSt^ points to the string entered by the user. The function returns True if the input is valid, or False if isn't. If the input is not valid, then ErrCode should be assigned an appropriate error code, and ErrorSt should be assigned the address of a static string containing a brief error message.

One thing a validation routine should *not* do is to modify EFP^.efEditSt^. Doing so can cause serious problems. People frequently try to do this when they're validating a field in an entry screen. What you should do instead is modify the variable associated with the field within a post-edit routine and then redraw the field. More generally, a validation routine has no business modifying any part of the field pointed to by EFP.

Now let's look at an actual validation routine, the standard one for fields of an integer type:

```
function ValidateLong(EFP : EntryFieldPtr; var ErrCode : Word;
                      var ErrorSt : StringPtr) : Boolean;
  {-Validate a field of type longint/word/integer/byte/shortint}
var
  L : LongInt;
  S : string[80];
begin
  ValidateLong := False;
  with EFP^ do begin
    StripPicture(efEditSt^, S);
    if not efStr2Long(S, L) then begin
      ErrCode := ecBadFormat;
      ErrorSt := @emInvalidNumber;
```

```
      end
      else if (L < efRangeLo.rtLong) or (L > efRangeHi.rtLong) then begin
        ErrCode := ecOutOfRange;
        ErrorSt := @emOutOfRange;
      end
      else
        ValidateLong := True;
    end;
  end;
```

If you compare this to the sample conversion routine in the previous section, you'll see that they are quite similar. The first step here is to strip the picture mask characters using StripPicture, yielding a "raw" string (S), which should represent a number. The next step is to try to convert that string to a LongInt using efStr2Long. If efStr2Long fails, the validation routine returns False and sets ErrCode to ecBadFormat, a constant declared in OPROOT, and points ErrorSt to the @emInvalidNumber message, also declared in OPROOT. If the conversion is successful, the next step is to determine whether or not the value is in the proper range: EFP^.efRangeLo.rtLong..EFP^.efRangeHi.rtLong. If it is, ValidateLong returns True. If it isn't, ValidateLong sets ErrCode to ecOutOfRange and ErrorSt to point to emOutOfRange, both of which are constants declared in OPROOT.

Although this is a typical validation routine for a numeric field, it is somewhat atypical of user-written validation routines, which tend to involve string fields rather than numeric fields. One common need, for example, is to verify that a string entered by the user appears within a list of acceptable responses. You'll find a good example of such a routine in ENTRY.PAS. You'll also find in ENTRY.PAS examples of custom validation routines that use the ValidateSubfields function documented later in this section.

Built-in Validation Routines

There are relatively few standard validation routines in OPFIELD, for two reasons. First, many field types (string, array of char, char, boolean, yes-no) normally don't require special validation, either because a picture mask alone can insure a valid response, or because there's no way for OPFIELD to know what is or isn't valid for such a field. Second, in the case of numeric field types, a single validation routine can often be used for multiple numeric types.

The list below briefly describes the standard validation routines. Note that several data types not mentioned here--date and time, for example--perform validation by overriding the Validate method rather than using a driver routine.

NullValidation
A do-nothing validation routine that always returns True.

{$IFDEF UseBCD}
ValidateBCD, SimpleValidateBCD
Validates a (simple) field of type BCD.

ValidateChar, SimpleValidateChar
Validates a (simple) field of type Char. This routine is used only if the Char field has a limited range, such as 'A'..'Z'; if it doesn't, NullValidation is used instead.

{$IFOPT N+}
ValidateExt, SimpleValidateExt
Validates a (simple) field of type Extended, Double, Single, or Comp.

ValidateLong, SimpleValidateLong
Validates a (simple) field of type LongInt, Word, Integer, Byte, or ShortInt.

ValidateReal, SimpleValidateReal
Validates a (simple) field of type Real.

In addition, OPFIELD also supplies three routines that are sometimes helpful in validating the contents of a (regular) string field. Two of them, ValidateNoBlanks and ValidateNotPartial, may be used as-is. The third, ValidSubfields, requires an extra parameter and must be called from within a user-written validation routine. These three routines are documented below.

ValidateNoBlanks / ValidateNotPartial

Declaration

```
function ValidateNoBlanks(EFP : EntryFieldPtr; var ErrCode : Word;
                          var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that no usable subfields in a string contain spaces.

Description

This validation routine verifies that a field is completely full. Like all high-level validation routines, it should never be called directly (i.e., it should only be called by a field editor). Note that use of this validation routine imposes more stringent requirements than designating a field as required, since required fields may contain partial entries. Note that ValidateNoBlanks may not be used with "simple" string fields.

Example

```
ES.AddStringField('State', 10, 18, 'AA', 10, 25, 2, 4, Scrap.State);
ES.ChangeValidation(MyFieldId, ValidateNoBlanks);
```

By specifying ValidateNoBlanks as the validation routine for this field, we insure both that the user will not leave the field blank and that he cannot enter a single letter for the state.

Declaration

```
function ValidateNotPartial(EFP : EntryFieldPtr; var ErrCode : Word;
                            var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that no usable subfields in a string contain spaces unless all do.

Description

This validation routine verifies that a field is either completely full or completely empty. Like all high-level validation routines, it should never be called directly. Note that ValidateNotPartial may not be used with "simple" string fields.

Example

```
ES.AddStringField('State', 10, 18, 'AA', 10, 25, 2, 4, Scrap.State);
ES.ChangeValidation(MyFieldId, ValidateNotPartial);
```

By specifying ValidateNotPartial as the validation routine for this field, we both insure that the user does not enter a single letter for the state and allow him to leave the field blank if the state is unknown.

Declaration

```
function ValidateSubfields(SubfieldMask : string;
                           EFP : EntryFieldPtr; var ErrCode : Word;
                           var ErrorSt : StringPtr) : Boolean;
```

Purpose

Validate that subfields in a string meet the requirements of SubfieldMask.

Description

This is a low-level routine that should be called only by a higher-level validation routine. It is intended to assist the programmer in writing custom validation routines for fields containing multiple subfields, some of which are required and some of which are not.

A SubfieldMask must be of the same length as the picture mask for the field being validated. The following characters are meaningful in a SubfieldMask:

```
r  (ReqdChar)       subfield is Required, no blank spaces allowed
u  (UnlessChar)     subfield is required, Unless entire field is empty
p  (PartialChar)    subfield is optional, but cannot be Partially full
```

Note that these characters may not be mixed within a given subfield.

See the demonstration program ENTRY.PAS for two typical examples of the use of ValidateSubfields.

Note too that ValidateSubFields may not be used with "simple" string fields.

Example

```
const
  PhoneMask   : string[14] = '(999) 999-9999';
  ValidPhone  : string[14] = '(ppp) uuu-uuuu';
...
{$F+}
function ValidatePhone(EFP : EntryFieldPtr; var ErrCode : Word;
                       var ErrorSt : StringPtr) : Boolean;
begin
  ValidatePhone := ValidateSubfields(ValidPhone, EFP, ErrCode, ErrorSt);
end;
{$F-}
```

Here ValidatePhone is a high-level validation routine that uses ValidateSubfields to verify that the last two subfields are not empty unless the entire field is empty, and that the first subfield is either completely full or completely empty.

See Also

ValidateNoBlanks ValidateNotPartial

7. Field Drawing Routines

One type of driver that you'll probably never need to write yourself is a draw routine. There are only two standard draw routines, and they are adequate to handle all the standard field types except the MultiLine field (which overrides the Draw method). Nevertheless, we'll give you a brief description of what a draw routine looks like and does, just in case. If you want to see an actual example of a draw routine, you should consult the source code for OPFIELD.

A draw routine has the following basic form:

```
procedure MyDrawProc(var Field;      St : string;     Row, Col : Word;
                     FA, CA : Byte;  POffset : Byte;  PasswordChar : Char;
                     Flags : PictureFlags);
var
  EF : EntryField absolute Field;
begin
  with EF do begin
    {draw St}
  end;
end;
```

The first parameter is the field record itself. It is declared as an untyped parameter because the compiler does not permit circular type references (that is, the type definition for EntryField includes a field of type DrawProc, so the type definition for a DrawProc may not refer to an EntryField). We therefore must declare an absolute variable of type EntryField on top of Field if we want to be able to treat it as an EntryField.

The next parameter is the string to be displayed. Contrary to what you might expect, a draw routine should not try to display EF.efEditSt^, because most of the field editors work with a copy of that string (to save time and code space) and don't update efEditSt^ until they have to. In any case, the St parameter contains the string that needs to be displayed. Note, however, that this string should not necessarily be displayed in its entirety. In some cases only part of the string being edited is displayed at any one time. EF.sfFWidth has the actual number of characters to be displayed.

The next two parameters, Row and Col, specify the absolute screen coordinates where St should be displayed. Although the EntryField does keep a record of the field's coordinates, those coordinates are relative to a particular location on the screen, and the Draw routine does not have the information necessary to translate them into absolute coordinates.

The FA and CA parameters contain the video attributes in which normal characters (FA) and control characters (CA) should be displayed. Although the EntryField stores this information too, it stores several pairs of attributes that are used in different situations. FA and CA specify the attributes to be used in the current situation. The CA parameter is used only when the efMapCtrls options is in effect for the field.

POffset is the horizontal offset for the string being displayed, which is usually 0. It is non-zero only if the draw routine is being called by a field editor that has scrolled the field horizontally. The general rule is this: The draw routine must display EF.sfFWidth characters from St starting at St[POffset]. If there are not EF.sfFWidth characters to display, the remainder of the field should (usually) be padded with efPadChar. efPadChar is also (usually) used to represent all blank spaces at the end of the field (or the beginning of the field, if it is right-justified).

PasswordChar is meaningful only if the expression

```
LongFlagIsSet(EF.sfOptions, efPasswordMode)
```

is True, indicating that password mode is in effect. If it is True, then all meaningful characters in St should be represented as PasswordChar.

The Flags parameter, an array[1..255] of Boolean, is needed only when password mode is in effect. It is used to identify literal and semi-literal characters that should never be represented with PasswordChar. Here, for example, is an imaginary string field containing a telephone number. Password mode is in effect, and PasswordChar is '*'. The user has filled in all but the last two spaces in the field. Each 't' (True) in the Flags field indicates a place where the user is allowed to enter text; each 'F' (False) indicates a character that may not be changed.

```
EF.efPicture^  '999-999-9999'
Flags           tttFtttFtttt
St             '408-555-12  '
Result         '***-***-**  '
```

If you've read through the lengthy list of options available for EntryFields, you're probably wondering if there's more involved in writing a draw routine than this. The answer is "Yes and no," because the answer depends on whether or not the draw routine in question aims to support all of the available field options, and a user-written draw routine likely would not. The issue is probably moot, in any case, because one of two built-in draw routines should suffice in almost every situation. If you need to write your own draw routine, our best advice is to make a copy of one of the standard draw routines and modify it: strip out what you don't need, and add in what it lacks.

Since you are far more likely to reuse one of the existing draw routines than to write one of your own, we've included the following table to show you which field options are supported by the built-in draw routines. For the sake of completeness, we've included the Draw method for the MultiLine field as well as the standard driver routines, DrawString and SimpleDrawString. The first section of the table shows the primary field options ('ef' prefix), and the second section shows the secondary options ('sef' prefix). The third section indicates whether or not the draw routine pays any attention to the pad character, password character, and picture mask.

Options	DrawString	SimpleDrawString	MultiLineField.Draw
efMapCtrls	Yes	Yes	Yes
efRightJustify	Yes	No	No
efPasswordMode	Yes	Yes	No
efParensForMinus	Yes	No	No
efHouseCursorAtEnd	No	Yes	No
sefBlankNotCurrent	Yes	Yes	Yes
sefPadCurrentOnly	Yes	Yes	Yes
Pad character	Yes	Yes	Yes
Password character	Yes	Yes	No
Picture mask/Flags	Yes	No	No

8. Stream Registration Routines

The stream registration routines for the various field types make it possible to store entry screens in stream files and libraries. Although it is unlikely that you'll ever need to write one of your own, we'll look briefly at a typical one to give you an idea of what they do. The one shown below is for a WordField that uses the "regular" editor.

```
procedure WordFieldStream(SPtr : IdStreamPtr);
  {-Register relevant types, procedure pointers}
begin
  EntryFieldStream(SPtr);

  with SPtr^ do begin
    RegisterType(otWordEField, veWordEField, TypeOf(WordField),
                 @WordField.Store, @WordField.Load);
    RegisterPointer(ptValidateLong, @ValidateLong);
    RegisterPointer(ptWordConversion, @WordConversion);
    RegisterPointer(ptDrawString, @DrawString);
    RegisterPointer(ptStringEditor, @StringEditor);
  end;
end;
```

The first thing to be done here is to register the parent type(s). Since a WordField is derived from the generic EntryField, we call EntryFieldStream, which registers both the EntryField and, by calling SelectFieldStream, the SelectField object types.

The next step is to register the WordField itself by calling the RegisterType method. This method takes five parameters: the code for the object type (otWordEField), its version number (veWordEField), a VMT pointer ('TypeOf(ObjectName)'), and the addresses of the Store and Load methods for the object.

The final step is to use RegisterPointer to register pointers to the standard driver routines used by a WordField. In this case, there are four such procedures to be registered: the validation routine, the conversion routine, the draw routine, and the field editor. If the field type relied entirely on methods to perform these functions, rather than driver routines, there would be no need to call RegisterPointer here. (The MultiLineField is an example of such a field.)

Here again, we can't pretend that this brief account is adequate to allow you to write your own registration routine. There are too many other steps involved. It is intended merely as a guide to understanding the numerous examples to be found in the source code for OPFIELD.

Built-in Registration Routines

The table below shows all the stream registration routines for specific field types in OPFIELD. The only other registration routines in OPFIELD are those for the lower-level field types: TextFieldStream (TextField), SelectFieldStream (SelectField), and EntryFieldStream (EntryField). Except in cases where you are developing your own objects derived from one of these field types, you should never need to call any of these three routines, because they are called for you automatically when you register higher level objects.

The column labels for the table have the following meanings. *Regular* is the registration routine normally used for a field with a picture mask. *Numeric* is the one used if the field is being edited with the NumberEditor. *Simple* is the one used if the field lacks a true picture mask. The name of the field type in question may be determined by removing "Stream" from the word in the first column ("ArrayFieldStream" - "Stream" = "ArrayField"). The names in parentheses

indicate the corresponding constructor. For example, if a ByteField was initialized by the InitNum constructor, the corresponding registration routine would be ByteFieldNumStream.

Regular (Init)	Numeric (InitNum)	Simple (InitSim)
ArrayFieldStream	---	---
BcdFieldStream	BcdFieldNumStream	BcdFieldSimStream
BooleanFieldStream	---	BooleanFieldSimStream
ByteFieldStream	ByteFieldNumStream	ByteFieldSimStream
CharFieldStream	---	CharFieldSimStream
ChoiceFieldStream	---	---
CompFieldStream	CompFieldNumStream	CompFieldSimStream
DateFieldStream	---	---
DateStFieldStream	---	---
DoubleFieldStream	DoubleFieldNumStream	DoubleFieldSimStream
ExtendedFieldStream	ExtendedFieldNumStream	ExtendedFieldSimStream
IntegerFieldStream	IntegerFieldNumStream	IntegerFieldSimStream
LongIntFieldStream	LongIntFieldNumStream	LongIntFieldSimStream
MultiLineFieldStream	---	---
NestedFieldStream	---	---
RealFieldStream	RealFieldNumStream	RealFieldSimStream
ShortIntFieldStream	ShortIntFieldNumStream	ShortIntFieldSimStream
SingleFieldStream	SingleFieldNumStream	SingleFieldSimStream
StringFieldStream	---	StringFieldSimStream
TimeFieldStream	---	---
WordFieldStream	WordFieldNumStream	WordFieldSimStream
YesNoFieldStream	---	YesNoFieldSimStream

D. OPFEDIT: Field Editors

This unit serves as a supplement to OPFIELD, and contains five driver routines--all of them field editors--that together are too large to fit in the crowded confines of OPFIELD. Unless you are working with derived field types, you will never need to refer to any of these routines directly, so we will again dispense with the usual detailed discussion of all interfaced routines in the unit and just give you a brief summary of the five field editors, with emphasis on the differences among them.

StringEditor

This is the most widely used of the five field editors. It is used for editing almost all of the field types supported by OPFIELD in cases where the field has a true picture mask (i.e., a "regular" field). Although it works with a wide variety of data types, it is oriented toward the editing of strings rather than numbers. Its most noteworthy feature is its ability to scroll a field horizontally to permit the editing of strings that are longer than the field itself. Its most noteworthy limitation is its handling of picture masks containing discretionary commas and floating dollar signs. When editing fields whose picture masks contain either of these characters, it must temporarily strip them out while editing, then put them back after the number has been entered.

SimpleStringEditor

This editor is used for entering all "simple" fields except those that are handled by SimpleCharEditor (discussed below). Although similar in many respects to StringEditor, it cannot handle fields with true picture masks. It assumes that all characters in the picture mask are the same as the first, and it uses this character only to determine which characters are acceptable and to adjust their case. Like StringEditor, it can scroll a field horizontally if necessary during editing.

NumberEditor

This special-purpose editor is designed exclusively for entering and editing numbers (Integer, Real, BCD). It works like a calculator, meaning that the cursor is always positioned at the far right edge of the field, and digits are always inserted at the end, pushing previously entered digits to the left. Since the cursor always sits in one place, cursor movement commands are always treated as exit commands: ccLeft, for example, is interpreted to mean "move the cursor to the field to the left." Unlike StringEditor, NumberEditor correctly displays discretionary commas and floating dollar signs as text is being entered, rather than inserting them after the fact. A field that uses this editor is referred to as a "numeric" field.

CharEditor

This editor is used for editing "regular" fields of type Char, Boolean, and YesNo. Although it does work with picture masks, it assumes that there is only one formatting character in the mask, and it always positions the cursor on the first one that it finds in the field. Like NumberEditor, it treats cursor movement commands as exit commands.

SimpleCharEditor

This editor is used for editing "simple" fields of type Char, Boolean, and YesNo. The principal difference between it and CharEditor is that it assumes the field is one character wide, so it pays attention only to the first character in the picture mask, just as SimpleStringEditor does.

Note that, in addition to these five driver routines, there are three other standard field editors that are implemented as methods: ChoiceField.Edit (multiple choice fields),

MultiLineField.Edit (multi-line fields), and PickField.Edit (pick list fields). These routines are discussed briefly in the documentation for OPFIELD and OPENTRY.

General Form of a Field Editor

As with the other three kinds of driver routines–conversion, validation, and draw--it is possible to write your own field editor. This section provides a brief discussion of the general form of such a routine. It is by no means a tutorial, however. It is intended only as a guide to understanding the source code for the five field editors implemented in OPFEDIT, which you should use as models if you ever need to write one yourself.

A field editing routine has the following basic form:

```
procedure MyEditProc(var Field;                Row, Col : Word;
                     FA, CA : Byte;            PasswordChar : Char;
                     PosCode : Byte;           ReadOnly : Boolean;
                     var CC : Word;            var ChWord : Word;
                     var InsertMode : Boolean; ErrorRoutine : ErrorProc;
                     UnitCode : Byte;          var CP : CommandProcessor);
var
  EF : EntryField absolute Field;
begin
  {...edit EF.efEditSt^...}
  with EF do begin
    {repeat}
      {redraw the field using the Draw method}
      {get next command}
      {process the command}
      {if exit command other than ccQuit issued then begin}
        {if not EF.Validate(...) then begin}
          {call ErrorRoutine}
          {don't exit}
        {end}
      {end}
    {until OK to exit}
  end;
end;
```

The first parameter is the field record itself. It is declared as an untyped parameter because the compiler does not permit circular type references (that is, the type definition for EntryField includes a field of type EditProc, so the type definition for an EditProc may not refer to an EntryField). We therefore must declare an absolute variable of type EntryField on top of Field if we want to be able to treat it as an EntryField.

The next two parameters, Row and Col, specify the absolute screen coordinates where the field should be displayed. Although the EntryField does keep a record of the field's coordinates, those coordinates are relative to a particular location on the screen, and the field editor does not have the information necessary to translate them into absolute coordinates.

The FA and CA parameters contain the video attributes in which normal characters (FA) and control characters (CA) should be displayed. Along with the PasswordChar parameter, these are generally not used directly by the field editor, but rather passed on to the field's Draw method.

PosCode is a value in the range 0..4, indicating where the cursor should be positioned initially. It will be non-zero only if the call is coming from an EntryScreen, in which case the value will depend on the command used to exit from the previous field and enter the current one. If

PosCode is 1 (last command = ccLeft or, in the case of multi-line fields, ccUp), the cursor should appear at the end of the field. If PosCode is 2 (last command = ccBackTab), the cursor should appear at the start of the last subfield. If PosCode is 3 (last command = ccWordLeft), the cursor should appear at the start of the last word in the field. If PosCode is 4, the cursor should appear at EF.sfFCPos, which contains the position of the cursor relative to the start of the field the last time that the field was edited. This code is used by an EntryScreen in cases where the user has issued a field movement command, such as ccUp, that had to be ignored, and the cursor had to be left on the current field. In all other cases, PosCode will be 0, meaning that the cursor should be placed either at the beginning of the field or, if the efCursorToEnd option is set for the field, just beyond the last character entered in the field. (Note that some or all of these position codes may be ignored by a given field editor. NumberEditor, for example, always places the cursor at the end of the field and leaves it there during editing.)

ReadOnly indicates that the editor is being called from an EntryScreen that is in read-only mode. If ReadOnly is True, and the user issues a command not in the set of ReadOnlyCommands, it should be ignored.

CC should be used to hold the commands entered by the user. It is made a var parameter so that the caller will know what command was used to exit from the field editor. Similarly, ChWord is the last key pressed by the user.

InsertMode indicates whether the field editor should be in insert or overtype mode initially. If the user changes modes during editing using the ccIns command, then the (undocumented but interfaced) ToggleInsertMode routine should be called to change the shape of the cursor, update the BIOS's status flag, and update InsertMode: 'ToggleInsertMode(InsertMode);'.

ErrorRoutine points to the routine to be called when an error occurs. It is normally used only to report errors returned by a field's Validate method, which always needs to be called before leaving the field unless the exit command is ccQuit.

UnitCode is the unit code for the caller. Besides being the first parameter in a call to ErrorRoutine, it is also passed to the help routine associated with the specified CommandProcessor, CP, when the ccHelp command is issued: 'CP.cpGetHelp(UnitCode, nil, EF.sfHelpIndex);'.

CP is the caller's CommandProcessor, which should be used to obtain commands from the user: 'CC := CP.GetCommand(ChWord);'.

E. OPSELECT: Field Selection

In the earlier sections of this chapter, we have dealt with fields as individual entities. In the next two sections, we'll be looking at higher level objects that work with groups of fields. Let's reconsider one of the hierarchy of objects mentioned in the introduction to the chapter; the three objects whose names are not surrounded by brackets are implemented in OPSELECT.

```
[CommandWindow]
  AbstractSelector
    Selector
      ScrollingSelector
    [EntryScreen]
      [ScrollingEntryScreen]
```

The AbstractSelector is an abstract object that simply defines some data fields and methods that can be shared in common by both the Selector and the EntryScreen, implemented in OPENTRY. Although the AbstractSelector introduces the concept of lists of fields, and provides numerous methods (mostly internal) for working with those lists, it does not provide any means for displaying the fields in a list. Such tasks are left to the higher level objects derived from it.

The Selector extends the AbstractSelector concept to provide methods for actually displaying the fields in the list and for processing commands that allow the user to move the cursor from field to field and to make selections.

The ScrollingSelector differs from the Selector in two important respects: it allows you to define a list of fields that cannot be displayed all at once within the selector's window, and it allows that window to be zoomed and resized. A non-scrolling Selector is constrained by the limits of its window.

1. The AbstractSelector

As indicated previously, the AbstractSelector is a purely abstract object; you'll never want to create an instance of one. If you are working with Selectors or EntryScreens, you will, however, need to be familiar with it because it defines some methods that are used frequently when working with objects derived from it.

Field Lists and Field IDs

Any object derived from an AbstractSelector maintains two field lists: one for user-selectable fields (SelectFields for Selectors and EntryFields for EntryScreens), and one for display-only fields (TextFields). In each case, fields are added to the list by calling a method with a name like AddXxxField. As fields are added to a list, they are automatically assigned ID numbers. The first field in a list is assigned an ID of 0, the next has an ID of 1, and so on. The ID number provides a convenient means of referring to a particular field. Both the AbstractSelector and the objects derived from it provide methods for modifying a particular field (changing its colors or option settings), and these methods always ask for the field's ID number as a parameter. For example, if you want to designate a certain field as protected after it has been added to the list, you would execute the statement

```
AS.ChangeProtection(5, True);
```

where AS is the name of your Selector/EntryScreen, and 5 is the ID number for the field to be protected.

Very important: Both the AbstractSelector and the objects derived from it assume that fields will be added in a particular order, going from left to right and from top to bottom. For example, if an EntryScreen has a 3x3 matrix of fields, the nine fields must be added in the order:

```
0____ 1____ 2____
3____ 4____ 5____
6____ 7____ 8____
```

Failure to add fields in the proper order may lead to unpredictable results. The reasons for this assumption will be explained in the discussion of "Field Movement Commands and Field Links," below.

Protected, Hidden, and Invisible Fields

In the simplest cases, all of the user-selectable fields in a Selector or EntryScreen are visible and accessible at all times. But if you work with these objects for any length of time, you will surely run into cases where you need to prevent the user from selecting/editing a particular field, if only temporarily, or where you need to hide it from view entirely. This is particularly true of data entry screens, which frequently need to display "calculated fields": protected fields whose values are calculated based on the values of one or more other, non-protected fields.

A *protected field* is a field whose contents are visible to the user, but which cannot be selected or modified by him. If he tries to move the cursor to a protected field, the cursor will move to the same field it would move to if the protected field didn't exist. A protected field is usually designated as such at the time it is added. In the case of a Selector, this is done by surrounding the call to add the field with the following statements:

```
Sel.slFieldOptionsOn(sfProtected);
Sel.AddField(...);
Sel.slFieldOptionsOff(sfProtected);
```

In the case of an EntryScreen, similarly, you would surround the call with these two statements:

```
ES.esFieldOptionsOn(efProtected);
ES.AddXxxField(...);
ES.esFieldOptionsOff(efProtected);
```

If you need to change the protection status of a field later, you can use the ChangeProtection method defined by the AbstractSelector. This method is typically called by a post-select (Selector) or post-edit (EntryScreen) routine in response to something entered by the user in another field.

A *hidden field* is similar to a protected field, except that the user cannot see it. Whereas protected fields are most often used for calculated fields, hidden fields are generally used when certain fields need to be displayed only in certain cases. For example, if a data entry screen is being used to obtain personnel information, it might hide certain fields if the user says an employee is single, and reveal them if the employee is married. Hidden fields, we should point out, are in fact displayed on the screen. But they are displayed in a video attribute that renders them effectively hidden. If the base text color for an entry screen is white on blue ($1F), for example, protected fields would be displayed in blue on blue ($11).

Hidden fields are created in the same way as protected fields, only the name of the field option is different:

```
Sel.slFieldOptionsOn(sfHidden);
Sel.AddField(...);
Sel.slFieldOptionsOff(sfHidden);
```

or

```
ES.esFieldOptionsOn(efHidden);
ES.AddXxxField(...);
ES.esFieldOptionsOff(efHidden);
```

To change the hidden status of a field, you would use ChangeHidden rather than ChangeProtection.

The difference between a hidden field and an *invisible field* is that an invisible field is not drawn on the screen at all. The display routines ignore such a field entirely. Although you might think that there is no real difference between a hidden field and an invisible field, there is actually a very big difference. Consider the following case. A data entry screen needs to use the same location on the screen for one of three fields, depending on the circumstances, and those circumstances can change dynamically based on the responses given by the user in other fields. Hidden fields won't work in this case, because hidden fields are only *effectively* hidden: if the second and third of the three fields are the hidden ones, they will overwrite the first field when the entry screen is redrawn. If they are designated as invisible, however, this won't happen, because they won't be drawn at all.

The only drawback to using invisible fields is that the window must be completely redrawn if the visibility status of a field changes (as the result of a call to ChangeInvisibility) after the window has been drawn once. In the case of a ScrollingSelector or ScrollingEntryScreen, this is accomplished very easily: all you do is call the ResetScreen procedure, discussed more fully below, to tell the object that its virtual screen needs to be redrawn. If the call to ChangeInvisibility was made from a post-select or post-edit routine, that's all you need to do; if the object is not currently processing commands, you would need to call the Draw method as well. If the Selector or EntryScreen is not scrollable, the window will need to be erased before it is redrawn. To avoid unnecessary screen activity, you will probably want to ask that the work be done on a virtual screen and then copied to the physical screen, like so:

```
ES.ChangeInvisibility(ID1, True);
ES.ChangeInvisibility(ID2, True);
ES.ChangeInvisibility(ID3, False);
...
ES.ActivateWrite;
ES.Erase;
ES.Draw;
ES.DeactivateWrite;
```

Presumably ID1, ID2, and ID3 refer to fields that share the same location; this would make the third field the only visible one.

Invisible fields can also, of course, be created at the time they are added by setting the appropriate field option:

```
Sel.slFieldOptionsOn(sfInvisible);
Sel.AddField(...);
Sel.slFieldOptionsOff(sfInvisible);
```

or

```
ES.esFieldOptionsOn(efInvisible);
ES.AddXxxField(...);
ES.esFieldOptionsOff(efInvisible);
```

Note that these three characteristics of a field are not mutually exclusive--a field can be designated as both protected and invisible, for example--but they do have priorities. An invisible field is treated as invisible even if it is also designated as hidden or protected; a hidden field is treated as hidden unless it is designated as invisible; a protected field is displayed only if it is not designated as invisible or hidden. Normally a field will have at most one of the three characteristics. But if it has two or more, keep in mind that changing one characteristic will not change the others. For example, if a field is designated as both protected and hidden (redundant, since hidden fields are also protected), the field will remain protected even if it is later unhidden.

Field Attributes

As you probably know by now, the user-selectable fields in a Selector or EntryScreen are displayed in different attributes in different situations. These are the factors that determine the attribute in which a given field will be displayed, in order of decreasing priority:

- invisible vs. visible
- color system vs. monochrome (or black & white)
- hidden vs. non-hidden
- protected vs. unprotected
- selected (current) vs. unselected

An invisible field will not be displayed at all, of course. In all other cases, the choice of attribute will be based in part on the display adapter in use (color or monochrome). If the field is hidden, the entire field, including the prompt, will be displayed in an attribute based on the background portion of the attribute for normal text within the window--e.g., if the attribute for normal text is white on black ($0F), a hidden field will be displayed in black on black ($00). If the field is merely protected, it will be displayed in the attributes appropriate for a protected field (the attribute for the prompt can be different from that for the rest of the field). And if the field is the current field--if it is "selected"--it will (or can) be displayed in different attributes than the other fields.

The following is a list of the various methods that can be used to set or change the attributes for fields:

Hidden Fields
RawWindow.SetTextAttr

Protected Fields
SetProtectedPromptAttr, ChangeProtectedPromptAttr
SetProtectedFieldAttr, ChangeProtectedFieldAttr

Selected Fields
SetSelectedPromptAttr, ChangeSelectedPromptAttr
SetSelectedFieldAttr, ChangeSelectedFieldAttr

Unselected Fields
SetPromptAttr, ChangePromptAttr
SetFieldAttr, ChangeFieldAttr

Other
SetCtrlAttr, ChangeCtrlAttr

Except for SetTextAttr, which affects all hidden fields, the methods whose names start with "Set" affect only those fields that are later added to the list. They change the default attributes. The methods whose names start with "Change" affect only individual fields (one of the parameters to these methods is an ID number).

Note that it is not always necessary to call these methods to set the attributes for the fields. On the contrary, it is rarely necessary to call any of them. One of the parameters to a constructor for a Selector/EntryScreen is a ColorSet, which will be used to establish the default attributes for the fields. If these defaults are acceptable for all the fields, then there will be no need to call any of the methods shown above.

The Current Field

The "current field" for a Selector or EntryScreen is usually distinguished from the other fields in one or both of two ways. In the case of a Selector, the current field is "highlighted"--displayed in a different attribute. When the user presses the <Enter> key to select the field, a user-written routine will be called to perform an appropriate action based on the selection. In the case of an EntryScreen, the current field is the one where the cursor is displayed; whether it is highlighted or not is up to you. The user does not need to press <Enter> to select the current field; he can start editing it as soon as the cursor is moved to it. In an EntryScreen, the current field is, in a sense, "pre-selected."

There are two methods defined by AbstractSelector that are relevant in this discussion. The first is GetCurrentID, which is used to determine which field is the current field. It is often used in conjunction with GetLastCommand in cases where user exit commands need to be processed. For example:

```
ES.Process;
case ES.GetLastCommand of
   ...
   ccUser0 : if ES.GetCurrentID = 7 then
               {do something special} ;
end;
```

In this case, "something special" happens if the user gives the ccUser0 command when field 7 of the EntryScreen is the current field. GetCurrentID is also frequently used in pre- and post-edit routines.

The other method, SetNextField, is used to force the cursor to be moved to a particular field. For example:

```
ES.SetNextField(3);
repeat
  ES.Process;
  ...
until Finished;
```

would request that field 3 be made the current field when the entry screen is displayed. Assuming that the entry screen had not been displayed previously, the current field would otherwise be set to field 0 (or the lowest numbered unprotected field, if field 0 were protected). SetNextField can also be used in pre- and post-edit routines (and pre- and post-select routines) to force the cursor to be moved to a particular field. For example, if a post-edit routine needed to force the EntryScreen to ignore a user's last command and move the cursor to a particular field, say field 5, it would do this:

```
ES.SetLastCommand(ccNone);
ES.SetNextField(5);
```

The call to SetLastCommand cancels the user's command, and the call to SetNextField tells the EntryScreen to make field 5 current.

Field Movement Commands and Field Links

As indicated above, fields must be added in left-to-right, top-to-bottom order. The reason for having the AbstractSelector and its descendants make this assumption is to allow certain field movement commands--notably ccUp and ccDown--to move the cursor in the direction requested by the user. Let's reconsider the case of the 3x3 matrix; we'll assume that the cursor is on field 1:

```
0____  1__x_  2____
3____  4____  5____
6____  7____  8____
```

In the data entry routines for most libraries, pressing the <Down> arrow in this case would move the cursor to field 2, the next field that was added. But with an AbstractSelector-based object, pressing the <Down> arrow here would move the cursor to field 4--the intuitively correct response. Now, if the fields could be added in any order whatsoever, implementing this behavior would be a nightmare--especially since protected, hidden, and invisible fields must be accounted for. The algorithm becomes much simpler, and executes many times faster, if we can assume that the fields were added in left-to-right, top-to-bottom order.

Two questions arise naturally here. First, what if you *want* <Down> to move the cursor to the next field (field 2 in our example)? You just reassign <Down> to the ccNextField command and, presumably, <Up> to the ccPrevField command. See the chapter on OPCMD for instructions on how to assign a key to a different command.

Second, what if you want to change the order in which fields are "visited"? Let's say, for example, that our 3x3 matrix represents an EntryScreen. If the cursor started on field 0 and you pressed <Enter> repeatedly, the cursor would be moved to field 1, then 2, then 3, and so on. How would you make it move instead to field 3, then 6, then 1 in a snaking fashion? You would do it by changing the "field links" for all the fields.

Each user-selectable field has a two field links, values that indicate the ID numbers for the next and previous fields. By default, both field links are set to $FFFF (BadFieldID), indicating that the Selector/EntryScreen should ignore the field link and base its decisions strictly on the order in which the fields were added to the list. But if you change the field links you can override the default behavior. To get the snaking effect described above, for example, you would set the field links as follows:

Field	Prev Link	Next Link
0	8	3
1	6	4
2	7	5
3	0	6
4	1	7
5	2	8
6	3	1
7	4	2
8	5	0

There are two methods provided for changing field links. SetAllFieldLinks can be used for setting the field links for all the fields at once. You provide it only the forward (Next) links; it figures out the backward (Prev) links for you automatically. (It would see, for example, that the forward link for field 5 was 8, and therefore set field 8's backward link to 5.) SetFieldLinks changes the field links for a particular field. See the entries for these two methods, below, for details.

Text Fields and Screen Update Routines

Returning again to our example of the 3x3 matrix, let's suppose that we wanted to display column labels above the first row of fields:

```
One   Two   Three
0___  1___  2___
3___  4___  5___
6___  7___  8___
```

In some cases it might be feasible to make "One," "Two," and "Three" the prompts for fields 0-2. If that trick is an option, and the fields are part of a scrollable Selector/EntryScreen, it may be the best option. Since the prompt is considered part of the field, even if it is on a separate row, every effort will be made to insure that the prompt for the current field is visible on the screen (if the window is made too small, it may not be possible).

But what would you do if the fields already have prompts and you needed to display column labels or other text within the window? You'd have two alternatives. The best alternative is usually to use text fields. In the case shown above, for example, you might do this:

```
AS.AddTextField('One',   1, 01);
AS.AddTextField('Two',   1, 08);
AS.AddTextField('Three', 1, 15);
```

or this:

```
AS.AddTextField('One    Two    Three', 1, 1);
```

Separate calls would be required, of course, if the text needed to be displayed in an attribute other than the default attribute for text in a window (in which case you would use AddTextFieldCustom or AddTextFieldDeluxe).

Another alternative would be to use a screen update routine. This hook allows you to specify a routine that is to be called each time that the inner portion of the window has to be redrawn. Using this hook, the case of the three column labels might be handled like this:

```
{$F+}
procedure MyUpdateProc(ASP : AbstractSelectorPtr);
begin
  with ASP^ do begin
    wFastText('One',   1, 01);
    wFastText('Two',   1, 08);
    wFastText('Three', 1, 15);
  end;
end;
{$F-}
...
AS.SetScreenUpdateProc(MyUpdateProc);
```

The advantage of this approach is added flexibility: the update routine can display text tied to the values of variables, for example. In most cases, however, text fields will do the job and should be considered the "cleaner" of the two alternatives.

We might add that the implementation of MyUpdateProc would have to change slightly if the Selector/EntryScreen was scrollable:

```
{$F+}
procedure MyUpdateProc(ASP : AbstractSelectorPtr);
begin
  FastText('One',   1, 01);
  FastText('Two',   1, 08);
  FastText('Three', 1, 15);
end;
{$F-}
```

The difference here is that the text has to be drawn to a virtual screen, rather than to the physical screen, so window-relative screen writing routines are inappropriate.

Manual Screen Updating

One important issue that we haven't touched on yet is that of forcing a Selector/EntryScreen to redraw a particular field or group of fields. There are only a few cases where this is necessary, but they arise fairly frequently. The most noteworthy one is the case of an EntryScreen that has a calculated field. Suppose, for example, that field 7's value is a function of the value of field 6, and that it needs to be recalculated and redisplayed each time the user changes field 6. The way to do this is to set up a post-edit routine that will check, after each field is edited, to see what the current field is. If GetCurrentID returns 6, it will recalculate the value of the variable associated with field 7, then call

```
ES.DrawField(7);
```

to redraw the field.

DrawField is the best choice when only a single field needs to be updated, but what if multiple fields need to be redrawn? We could call DrawField for each of them, of course, but the simplest way is to call ResetScreen once. This sets an internal flag that tells the Selector/EntryScreen that it needs to completely redraw the inner portion of its window.

Wrap Modes

Most of the options that affect the behavior of AbstractSelector-based objects are specified by calling methods implemented by the higher level objects. One option, however, is selected by calling a method implemented by the AbstractSelector. SetWrapMode allows you to specify what happens to the cursor when the user tries to move it off the edge of the window--e.g., if the cursor is on the top row and the user presses <Up>. If the WrapMode is *StopAtEdges*, the cursor stays where it is. If the WrapMode is *WrapAtEdges*, the cursor "wraps" around to the top or bottom row of the window. If the WrapMode is *ExitAtEdges*, the selector exits with an exit command of ccExitAtTop or ccExitAtBot, depending on the direction in which the cursor was being moved.

WrapAtEdges, the default setting, is usually best for non-scrollable selectors and entry screens. StopAtEdges is usually best for scrollable ones. ExitAtEdges is appropriate only in rare cases, such as the nested entry screen in ENTRY.PAS.

Constants

```
DefWrapMode : WrapMode = WrapAtEdges;
```

Default wrap mode for an object derived from an AbstractSelector.

```
On  = True;
Off = False;
```

These two constants are intended to help you write self-documenting code with OPSELECT and OPENTRY. They are typically used as parameters to methods such as ChangeProtection that turn certain options on or off.

Types

```
AbstractSelectorPtr = ^AbstractSelector;
AbstractSelector =
  object(CommandWindow)
    ...
  end;
```

Abstract object from which Selector and EntryScreen are derived.

```
asFieldProc = procedure(SFP : SelectFieldPtr; var D;
                        ASP : AbstractSelectorPtr);
```

A user-written procedure called by VisitAllSelectFields to perform a particular action on all SelectFields associated with a Selector.

```
asTextFieldProc = procedure(TFP : TextFieldPtr; var D;
                            ASP : AbstractSelectorPtr);
```

A user-written procedure called by VisitAllTextFields to perform a particular action on all TextFields associated with a Selector or EntryScreen.

```
ScreenUpdateProc = procedure(ASP : AbstractSelectorPtr);
```

A user-written procedure called each time the interior of an AbstractSelector's window is completely redrawn.

```
WrapMode = (StopAtEdges, WrapAtEdges, ExitAtEdges);
```

Type used to describe different modes of behavior in cases where the cursor is at the top or bottom row and the user tries to move the cursor to a non-existent field (for example, if the current field is on the top row, and the user presses <Up>). If the WrapMode is StopAtEdges, the cursor stays where it is. If the WrapMode is WrapAtEdges, the cursor "wraps" around to the top or bottom row of the window. If the WrapMode is ExitAtEdges, the selector exits with an exit command of ccExitAtTop or ccExitAtBot, depending on the direction in which the cursor was being moved.

AddTextField / AddTextFieldCustom

Declaration
```
procedure AddTextField(St : string; Row, Col : Word);
```

Purpose
Add a text field.

Description
This method adds a text field to a Selector or EntryScreen. The text (St) of the field will be displayed at the specified Row and Column, relative to the top left corner of the window, in the default video attribute for the window (TextColor or TextMono).

Example
```
AS.AddTextField('-Fixed text field-', 01, 4);
AS.AddTextField('-Fixed text field-', 10, 4);
```
Adds text fields at rows 1 and 10 of a selector window.

See Also
AddTextFieldDeluxe SetScreenUpdateProc

Declaration
```
procedure AddTextFieldCustom(St : string; Row, Col : Word; Color, Mono : Byte);
```

Purpose
Add a text field with specific colors.

Description
This method is identical to AddTextField, except that it allows you to specify the video attributes that it will be displayed in on Color and Monochrome systems.

Example
```
AS.AddTextFieldCustom('-Fixed text field-', 1, 4, $03, $70);
```
Adds a text string to the selector window. The string will be displayed in cyan on black for color monitors, reverse video for monochrome monitors.

See Also
AddTextFieldDeluxe SetScreenUpdateProc

Declaration

```
procedure AddTextFieldDeluxe(St : string; Row, Col : Word;
                             var ColorAttrs, MonoAttrs : FlexAttrs);
```

Purpose

Add a text field with flexible colors.

Description

This method differs from AddTextField and AddTextFieldCustom in that it allows you to create text fields that can be displayed in multiple colors, not just one. The low-level routine used to draw the text field is the FlexWrite procedure in OPCRT. The ColorAttrs and MonoAttrs parameters are used to specify the attribute array that will be passed to FlexWrite when running on color and monochrome systems, respectively. The St parameter is a string of the same form as one passed to FlexWrite. See the documentation for FlexWrite in the OPCRT chapter for further details. Note that SetScreenUpdateProc provides a good alternative to text fields in cases where the text being displayed needs to change at runtime.

Example

```
const
  ColorFlex : FlexAttrs = ($07, $0A, $0B, $0E);
  MonoFlex : FlexAttrs = ($07, $70, $07, $07);
...
  AS.AddTextFieldDeluxe(^A'Fancy '^A' '^B'Flex'^B' '^C'Text'^C, 1, 4,
                        ColorFlex, MonoFlex);
```

Uses flex writing to display a single text string in multiple colors.

See Also

AddTextField
AddTextFieldCustom
FlexWrite (OPCRT)
SetScreenUpdateProc

Declaration

```
procedure ChangeCtrlAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for control characters.

Description

This method allows you to change the attributes in which control characters in the specified field (ID) are displayed on Color and Monochrome systems.

Example

```
AS.ChangeCtrlAttr(5, $0A, $70);
```

Changes the control character attribute of field 5 to light green on black for color monitors, reverse video for monochrome monitors.

See Also

SetCtrlAttr

ChangeFieldAttr / ChangeHidden / ChangeInvisibility

Declaration
```
procedure ChangeFieldAttr(ID : Word; Color, Mono : Byte);
```

Purpose
Change attributes for unselected fields.

Description
This method allows you to change the attributes in which the specified field (ID) is displayed on Color and Monochrome systems when the field is unselected (i.e., it is not the current field).

Example
```
AS.ChangeFieldAttr(5, $07, $07);
```
Changes the normal field attribute of field 5 to light gray on black for color monitors, normal video for monochrome monitors.

See Also
SetFieldAttr

Declaration
```
procedure ChangeHidden(ID : Word; On : Boolean);
```

Purpose
Change the hidden status of the specified field.

Description
This method allows you to change the hidden status for the specified field (ID). If On is True, the field will become hidden (i.e., not visible, effectively protected). If On is False, the field will become unhidden (visible), but not necessarily unprotected (if the field was previously designated as both hidden and protected, it will remain protected). See the discussion of protected, hidden, and invisible fields at the beginning of this section for further details.

Example
```
AS.ChangeHidden(3, True);
```
Makes field 3 (the fourth field added) a hidden field.

See Also
ChangeProtection

Declaration
```
procedure ChangeInvisibility(ID : Word; On : Boolean);
```

Purpose
Change the invisibility status of the specified field.

Description
This method allows you to change the invisibility status for the specified field (ID). If On is True, the field will not be drawn at all the next time the screen is redrawn, and it will be effectively protected. If On is False, the field will be designated as "visible," although it will not necessarily be visible to the user (since it may also have been designated as hidden) or unprotected (if the field was previously designated as both invisible and protected, it will remain protected). See the discussion of protected, hidden, and invisible fields at the beginning of this section for further details.

Example
```
AS.ChangeInvisibility(3, True);
```
Makes field 3 (the fourth field added) an invisible field.

Declaration

```
procedure ChangePromptAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for prompts.

Description

This method allows you to change the attributes in which the prompt for the specified field (ID) is displayed on Color and Monochrome systems when the field is unselected (i.e., it is not the current field).

Example

```
AS.ChangePromptAttr(5, $02, $07);
```

Changes the prompt attribute of field 5 to green on black for color monitors, normal video for monochrome monitors.

See Also

SetPromptAttr

Declaration

```
procedure ChangeProtectedFieldAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for protected fields.

Description

This method allows you to change the attributes in which the specified field (ID) is displayed on Color and Monochrome systems when the field is protected.

Example

```
AS.ChangeProtectedFieldAttr(5, $07, $07);
```

Changes the protected field attribute of field 5 to light gray on black for color monitors, normal video for monochrome monitors.

See Also

SetProtectedFieldAttr

ChangeProtectedPromptAttr / ChangeProtection

Declaration

```
procedure ChangeProtectedPromptAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for protected prompts.

Description

This method allows you to change the attributes in which the prompt for the specified field (ID) is displayed on Color and Monochrome systems when the field is protected.

Example

```
AS.ChangeProtectedPromptAttr(5, $07, $07);
```

Changes the protected prompt attribute of field 5 to light gray on black for color monitors, normal video for monochrome monitors.

See Also

SetProtectedPromptAttr

Declaration

```
procedure ChangeProtection(ID : Word; On : Boolean);
```

Purpose

Change the protection status of the specified field.

Description

This method allows you to change the protection status for the specified field (ID). If On is True, the user will not be able to move the cursor onto the field or alter its contents. If On is False, the field will become accessible again, provided that the field was not already designated as hidden or invisible. See the discussion of protected, hidden, and invisible fields at the beginning of this section for further details.

Example

```
AS.ChangeProtection(5, True);
```

Turns on protection for field 5. See the example for SetAllFieldLinks for a complete program using ChangeProtection.

See Also

ChangeHidden

ChangeInvisibility

ChangeSelectedFieldAttr / ChangeSelectedPromptAttr

Declaration

```
procedure ChangeSelectedFieldAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for selected fields.

Description

This method allows you to change the attributes in which the specified field (ID) is displayed on Color and Monochrome systems when the field is selected (i.e., it is the current field).

Example

```
AS.ChangeSelectedFieldAttr(5, $03, $70);
```

Changes the selected field attribute of field 5 to cyan on black for color monitors, reverse video for monochrome monitors.

See Also

SetSelectedFieldAttr

Declaration

```
procedure ChangeSelectedPromptAttr(ID : Word; Color, Mono : Byte);
```

Purpose

Change attributes for selected prompts.

Description

This method allows you to change the attributes in which the prompt for the specified field (ID) is displayed on Color and Monochrome systems when the field is selected (i.e., it is the current field).

Example

```
AS.ChangeSelectedPromptAttr(5, $0F, $70);
```

Changes the selected prompt attribute of field 5 to white on black for color monitors, reverse video for monochrome monitors.

See Also

SetSelectedPromptAttr

Done / DrawField / FindField

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate field list, screen buffers.

Description
This destructor disposes of the field list associated with the Selector/EntryScreen, then calls the parent's destructor (CommandWindow.Done, which is actually RawWindow.Done) to dispose of screen buffers and other window-related data structures.

Example
```
AS.Erase;
AS.Done;
```
Erase AS and release all heap space.

See Also
RawWindow.Done

Declaration
```
procedure DrawField(ID : Word);
```

Purpose
Draw the specified field.

Description
This method redraws the field specified by ID. The most common uses for it are in post-edit routines and IncChoice procedures associated with EntryScreens, where it can be used to redraw a calculated field. See the discussion of calculated fields in the documentation for OPENTRY.

See Also
ResetScreen

Declaration
```
function FindField(ID : Word) : SelectFieldPtr;
```

Purpose
Return a pointer to the specified field.

Description
If successful, this method returns a pointer to the field designated by ID. If not, it returns nil. Although it is widely used by other methods, particularly those that modify individual fields, FindField is generally not needed by user programs, since higher level methods are available for changing most of the characteristics of a field after it is has been created.

Example
```
SFP := ES.FindField(5);
if SFP <> nil then
  EntryFieldPtr(SFP)^.efOptionsOn(efRequired);
```
Mark field 5 of an EntryScreen (ES) as a required field. (This particular task could also be performed by calling EntryScreen.ChangeRequired.) Notice that the pointer returned by FindField must be typecast to an EntryFieldPtr in order to treat the field it points to as an EntryField.

See Also
FindTextField

Declaration

```
function FindTextField(ID : Word) : TextFieldPtr;
```

Purpose

Return a pointer to the specified text field.

Description

This method allows to locate the text field specified by ID. It is provided to allow you to handle those rare cases where a text field needs to be modified dynamically. The first example shown below demonstrates, for example, how you could hide a text field (there is currently no provision made for hidden text fields). You could also use it to locate a text field whose text needs to be changed in certain situations, as shown in the second example.

Examples

```
TFP := AS.FindTextField(5);
if TFP <> nil then
  with TFP^ do begin
    FillChar(tfColorAttrs, SizeOf(tfColorAttrs), $11);
    FillChar(tfMonoAttrs, SizeOf(tfMonoAttrs), $00);
  end;
```

Assuming that the background color for AS is blue (1) on color systems and black (0) on mono systems, this would effectively hide text field 5 (the sixth field added) of AS.

```
TFP := AS.FindTextField(3);
if TFP <> nil then begin
  MyStringPtr := StringToHeap('New text for field 3');
  if MyStringPtr <> nil then begin
    DisposeString(TFP^.tfString);
    TFP^.tfString := MyStringPtr;
  end;
end;
```

This would change the text for field 3 to 'New text for field 3'. We don't recommend doing this, but if you need to do it, the code shown above demonstrates the proper technique. (StringToHeap and DisposeString are in OPSTRING.)

See Also

FindField

Declaration

```
function GetCurrentID : Word;
```

Purpose

Get the ID for the current field.

Description

This method returns the ID number for the current field. A typical application for it is in a pre-edit or post-edit routine for an EntryScreen, which might need to perform different tasks depending on what field is current.

Example

```
CurID := AS.GetCurrentID;
```

Returns the ID of the current field.

GetHelpIndex / InitCustom

Declaration

```
function GetHelpIndex(ID : Word) : Word;
```

Purpose

Get the help index for the specified field.

Description

This method returns the help index for the field specified by ID. It is most useful in Action routines, such as the one shown below.

Example

```
procedure Action(SP : SelectorPtr; ID : Word);
var LE : LineEditor;
begin
  LE.Init(DefaultColorSet);
  LE.SetHelpIndex(SP^.GetHelpIndex(ID));
  ...
end;
```

This Action routine (for a Selector) sets the help index for the line editor to that of the field being edited.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt;     var CP : CommandProcessor;
                       UnitCode : Byte
```

Purpose

Initialize an abstract selector with custom window options

Description

This constructor instantiates an AbstractSelector using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window. CP is the CommandProcessor to be used, and UnitCode is the unit code to be passed to error handlers and GetHelp routines. InitCustom will fail only if the parent constructor (AbstractSelector.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

This constructor is intended strictly for internal use. It is documented here only because Selector.InitCustom and EntryScreen.InitCustom both depend on it, and their behavior cannot be explained satisfactorily if this method is not documented.

InitCustom performs three important tasks. First, it initializes the lists of fields (either SelectFields or EntryFields) and text fields. Second, it calls CommandWindow.InitCustom to allocate screen buffers and perform other initialization tasks common to all CommandWindow-based objects. Third, it initializes all other data fields common to both Selectors and EntryScreens.

The most noteworthy of these is a field (asColors) that contains a copy of the ColorSet passed to InitCustom as the Colors parameter. There are seven pairs of attributes in a ColorSet that have special importance for any object derived from an AbstractSelector, all of them pertaining to fields: CtrlColor/Mono, PromptColor/Mono, SelPromptColor/Mono, ProPromptColor/Mono, FieldColor/Mono, SelFieldColor/Mono, and ProFieldColor/Mono. An EntryScreen makes special use of three other pairs of attributes, all pertaining to field delimiters: DelimColor/Mono, SelDelimColor/Mono, and ProDelimColor/Mono. For each of these pairs of attributes, there is a corresponding SetXxxAttr method that alters the appropriate fields in asColors. (The methods for changing attributes pertaining to delimiters are declared in EntryScreen, the others in AbstractSelector.)

See Also

CommandWindow.InitCustom

Declaration
```
procedure PositionCursorAt(SFP : SelectFieldPtr); virtual;
```

Purpose
Puts the cursor on the beginning of the specified field.

Description
This method does two things: it insures that the specified field is visible (if a ScrollingSelector or ScrollingEntryScreen is in use), and it positions the cursor at the start of the field.

Important: This method is intended primarily for internal use and contains no error checking. It assumes that SFP is a valid pointer to a field and that the window is active and current.

Example
```
AS.PositionCursorAt(AS.FindField(20));
```
Moves the cursor to the start of field 20 (whether it's currently on the screen or not).

See Also
FindField

Declaration
```
procedure ResetScreen;
```

Purpose
Call when contents of field(s) changed without calling DrawField.

Description
This method sets a flag for the Selector/EntryScreen telling it that the contents of its window needs to be redrawn. It is most commonly used in a post-edit routine for an EntryScreen that needs to change the colors or contents of several fields at a time. Rather than calling DrawField once for each field changed, you can call ResetScreen once to tell the object that it needs to redraw itself.

Note that it is not necessary to call ResetScreen following calls to ChangeHidden, ChangeProtection, or ChangeInvisibility. Those methods insure that the screen will be redrawn at the next opportunity.

Example
```
for I := 0 to 20 do
  AS.ChangePromptAttr(I, $0E, $0F);
AS.ResetScreen;
```
Change the prompt attribute for fields 0-20, and indicate that the window needs to be redrawn.

See Also
DrawField

SetAllFieldLinks

Declaration

```
procedure SetAllFieldLinks(var LinksMap);
```

Purpose

Set the forward and backward links for all fields.

Description

This method allows you to modify the order in which fields are "visited" as you move through a Selector/EntryScreen by specifying an explicit "forward link" for each field. (The forward link is the field the cursor is moved to when <Enter> or <Tab> is pressed.) This method must not be called until *after* all fields have been added to the selector/entry screen.

The variable passed as the LinksMap parameter should be an 'array[0..n] of Word', where *n* is the ID number of the last field. Each element in the array specifies the forward link for that field: LinksMap[0] has the forward link for field 0, LinksMap[1] has the forward link for field 1, and so on. Note that, given this list of forward links, SetAllFieldLinks will automatically calculate the appropriate backward links to insure logical behavior when the cursor is being moved backward (with <ShiftTab>, for example).

If a particular field is protected, you should set its forward link to 'BadFieldId' (a constant declared in OPFIELD) *unless* that field will later be unprotected. The Process method will ignore any forward or backward link that points to a protected field and revert to the default behavior of moving the cursor to the next/previous unprotected field added to the list.

When setting the backward links for all the fields, SetAllFieldLinks assumes that each unprotected field in the entry screen is named as the forward link for one of the other fields, and that only one field names a given field as its forward link. Specifically, when setting the backward link for field *N*, it scans the list of forward links looking for another field that points to *N*. The first one it finds becomes the backward link for field *N*. If none is found, the backward link is set to BadFieldID, indicating that Process should apply its usual set of rules for determining where to move the cursor. In general, then, it is best to use SetAllFieldLinks to establish a perfect loop linking all unprotected fields in the entry screen together (see the example below), then use SetFieldLinks to make any minor changes to individual links (to prevent wrapping at the edges of an entry screen, for example, which can be done by setting a field's forward or backward link to point to itself).

Calls to SetAllFieldLinks have no effect on the behavior of the ccUp and ccDown commands, which always move the cursor to the field immediately above or below the current field. If you want <Up> and <Down> to move the cursor to the previous/next field, you can reassign them to the ccPrevField and ccNextField commands, respectively.

Because field movement in a Selector is usually done with <Up> and <Down>, SetAllFieldLinks is generally more useful for an EntryScreen than a Selector. You will probably want to use it with a Selector only in cases where either 1) you have reassigned <Up> and <Down> to ccPrevField and ccNextField, or 2) enabled the slAutoSelect option, as demonstrated in the example program at the beginning of the next section.

Example

```
const                          {field: 0  1  2  3  4  5}
  LinksMap : array[0..5] of Word = (1, 3, 0, 4, 2, BadFieldId);
var  Sel  : Selector;
     Data : array[0..5] of String[10];
     I    : Word;
begin
  ...
  for I := 1 to 6 do
    Sel.AddField(I+2, 1, 10, I+2, 12, 10, 0);
  Sel.ChangeProtection(5, True);
  Sel.SetAllFieldLinks(LinksMap);
  Sel.SetNextField(2);
  ...
end.
```

Here the selector is set up such that the cursor starts at field 2 (because of the call to SetNextField), then moves to fields 0, 1, 3, 4 (and then back to 2), in that order, as <Tab> is pressed. Field 5 is a protected field, so no forward link is needed for it.

See Also

SetFieldLinks SetNextField

Declaration

```
procedure SetCtrlAttr(Color, Mono : Byte);
```

Purpose

Set attributes for control characters.

Description

This method allows you to specify the attributes in which the control characters within fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetCtrlAttr($0A, $70);
```

Sets the control character attribute to light green on black for color monitors, reverse video for monochrome monitors.

See Also

ChangeCtrlAttr

Declaration

```
procedure SetFieldAttr(Color, Mono : Byte);
```

Purpose

Set attributes for unselected fields.

Description

This method allows you to specify the attributes in which unselected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetFieldAttr($07, $07);
```

Sets the normal field attribute to light gray on black for color monitors, normal video for monochrome monitors.

See Also

ChangeFieldAttr

SetFieldLinks / SetNextField

Declaration

```
procedure SetFieldLinks(ID, Next, Prev : Word);
```

Purpose

Specify the fields to jump to when <Tab> or <ShTab> pressed on a given field.

Description

This method is similar to SetAllFieldLinks, except that it changes the forward (Next) and backward (Prev) links for a single field (ID) rather than for all fields. It may be called at any time after the field specified by ID has been added to the entry screen, even if the fields denoted by Next and Prev have not been added yet.

ID is the field whose forward and/or backward links you want to change. Next is the ID for the field to jump to when <Enter> is pressed (or any other event that causes the cursor to be moved to the next field). Prev is the ID for the field to jump to when <ShiftTab> is pressed (or any other event that causes the cursor to be moved to the previous field). If you want to override only the forward link, you can specify 'BadFieldId' (= $FFFF) as the Prev parameter. To override only the backward link, specify BadFieldId as the Next parameter. (When a forward or backward link is BadFieldId, Process will simply apply its usual set of rules for determining where to move the cursor.

Although SetAllFieldLinks is easier to use when numerous changes must be made, SetFieldLinks provides greater flexibility. Multiple fields may safely be given the same forward link, for example, and individual links may be changed easily at any time—even from within a post-edit routine.

Example

```
Sel.SetFieldLinks(0, 1, 2);
Sel.SetFieldLinks(1, 3, 0);
Sel.SetFieldLinks(2, 0, 4);
Sel.SetFieldLinks(3, 4, 1);
Sel.SetFieldLinks(4, 2, 3);
```

This does the same thing as the one call to SetAllFieldLinks in the example for that method.

See Also

SetAllFieldLinks

Declaration

```
procedure SetNextField(ID : Word);
```

Purpose

Set field to move cursor to next.

Description

This method allows you to specify a field to move the cursor to. There are three places in a program from which this method may safely be called: 1) just prior to a call to Process (the most common case), 2) within a pre-edit or pre-select routine, or 3) within a post-edit or post-select routine.

Example

See the example for SetAllFieldLinks.

See Also

SetAllFieldLinks

Declaration

```
procedure SetPromptAttr(Color, Mono : Byte);
```

Purpose

Set attributes for prompts.

Description

This method allows you to specify the attributes in which prompts for unselected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetPromptAttr($02, $07);
```

Sets the prompt attribute to green on black for color monitors, normal video for monochrome monitors.

See Also

ChangePromptAttr

Declaration

```
procedure SetProtectedFieldAttr(Color, Mono : Byte);
```

Purpose

Set attributes for protected fields.

Description

This method allows you to specify the attributes in which protected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetProtectedField($07, $07);
```

Sets the protected field attribute to light gray on black for color monitors, normal video for monochrome monitors.

See Also

ChangeProtectedFieldAttr

SetProtectedPromptAttr / SetScreenUpdateProc

Declaration

```
procedure SetProtectedPromptAttr(Color, Mono : Byte);
```

Purpose

Set attributes for protected prompts.

Description

This method allows you to specify the attributes in which the prompts for protected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetProtectedPromptAttr($07, $07);
```

Sets the protected prompt attribute to light gray on black for color monitors, normal video for monochrome monitors.

See Also

ChangeProtectedPromptAttr

Declaration

```
procedure SetScreenUpdateProc(SUP : ScreenUpdateProc);
```

Purpose

Set procedure to be called after a complete screen update.

Description

This method allows you to specify a routine that is to be called each time a Selector or EntryScreen is redrawn entirely. The main purpose of this hook is to provide an alternative to text fields in cases where ordinary text needs to be displayed within the window.

The update procedure should be of the following form:

```
{$F+}
procedure Update(ASP : AbstractSelectorPtr);
begin
  {...write to screen...}
end;
```

ASP points to the Selector/EntryScreen being updated. You may typecast this pointer to the appropriate type--e.g., 'EntryScreenPtr(ASP)'--if you need access to data or methods not available from an AbstractSelector.

Note that, if the Selector/EntryScreen is scrollable, then its virtual screen will be active at the time your update procedure is called. You may write text to the virtual screen using FastWrite or any of its kin. For example,

```
FastWrite('------ Divider ------', 75, 20, $07);
```

would write the specified text to row 75 of the virtual screen.

If the Selector/EntryScreen is not scrollable, your update procedure will be writing to the physical screen, rather than a virtual screen.

Example

```
{$F+}
procedure Update(ASP : AbstractSelectorPtr);
begin
  ASP^.wFastCenter('Fixed Header', 1, TextAttr);
end;
...
AS.SetScreenUpdateProc(Update);
```

Update simply adds a fixed heading inside the selector window.

Declaration

```
procedure SetSelectedFieldAttr(Color, Mono : Byte);
```

Purpose

Set attributes for selected fields.

Description

This method allows you to specify the attributes in which selected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetSelectedFieldAttr($03, $70);
```

Sets the selected field attribute to cyan on black for color monitors, reverse video for monochrome monitors.

See Also

ChangeSelectedFieldAttr

Declaration

```
procedure SetSelectedPromptAttr(Color, Mono : Byte);
```

Purpose

Set attributes for selected prompts.

Description

This method allows you to specify the attributes in which the prompts for selected fields are displayed on Color and Monochrome systems. Note that it does not affect fields already added to the Selector/EntryScreen, only those added subsequently.

Example

```
AS.SetSelectedPromptAttr($0F, $07);
```

Sets the selected prompt attribute to white on black for color monitors, normal video for monochrome monitors.

See Also

ChangeSelectedPromptAttr

SetWrapMode / VisitAllSelectFields

Declaration

```
procedure SetWrapMode(WM : WrapMode);
```

Purpose

Set the wrap mode for the selector.

Description

This method allows you to indicate how the Selector/EntryScreen should behave in cases where the cursor is at the top or bottom row and the user tries to move the cursor to a non-existent field (for example, the current field is on the top row, and the user presses <Up>). If WM is StopAtEdges, the cursor will stay where it is. If WM is WrapAtEdges, the cursor will "wrap" around to the top or bottom row of the window. If WM is ExitAtEdges, Process will exit with an exit command of ccExitAtTop or ccExitAtBot, depending on the direction in which the cursor was being moved. The default mode is WrapAtEdges.

Example

```
AS.SetWrapMode(StopAtEdges);
```

AS will ignore commands to go beyond the top or bottom rows.

Declaration

```
procedure VisitAllSelectFields(FProc : asFieldProc; var D);
```

Purpose

Call the specified procedure for all fields in the selector.

Description

This method allows you to specify a user-written procedure to be called for each SelectField associated with a Selector. FProc is the procedure to be called, and D is the data to be passed to it.

FProc must be a procedure of the form:

```
{$F+}
procedure FieldProc(SFP : SelectFieldPtr; var D; ASP : AbstractSelectorPtr);
begin
end;
```

SFP is a pointer to the SelectField to be modified. D is the data passed to VisitAllSelectFields. And ASP is a pointer to the selector itself.

VisitAllSelectFields is a low-level routine and must be used with caution. It is always safe to change video attributes, as shown in the example below. But global changes to option settings should generally be avoided.

Example

```
const BothAttr : Word = $0B70;
...
  {$F+}
  procedure GlobalPromptAttr(SFP : SelectFieldPtr; var D;
                             ASP : AbstractSelectorPtr);
  var W : Word absolute D;
  begin
    SFP^.SetPromptAttr(Hi(W), Lo(W));
  end;
  {$F-}
...
AS.VisitAllSelectFields(GlobalPromptAttr, BothAttr);
```

"Visits" all selector fields with the procedure GlobalPromptAttr.

See Also

EntryScreen.VisitAllEntryFields VisitAllTextFields

Declaration

```
procedure VisitAllTextFields(FProc : asTextFieldProc; var D);
```

Purpose

Call the specified procedure for all text fields in the selector.

Description

This method allows you to specify a user-written procedure to be called for each TextField associated with a Selector or EntryScreen. FProc is the procedure to be called, and D is the data to be passed to it.

FProc must be a procedure of the form:

```
{$F+}
procedure TextFieldProc(TFP : TextFieldPtr; var D; ASP : AbstractSelectorPtr);
begin
end;
```

TFP is a pointer to the TextField to be modified. D is the data passed to VisitAllTextFields. And ASP is a pointer to the selector itself.

VisitAllTextFields should be used only to make global changes to video attributes, as shown in the example below.

Example

```
const
  NewColorFlex : FlexAttrs = ($0A, $07, $0B, $0E);
...
  {$F+}
  procedure GlobalFlexAttr(TFP : TextFieldPtr; var D;
                           ASP : AbstractSelectorPtr);
  begin
    TFP^.tfColorAttrs := FlexAttrs(D);
  end;
  {$F-}
...
AS.VisitAllTextFields(GlobalFlexAttr, NewColorFlex);
```

Changes the color flex attributes for all text fields by "visiting" each field with the procedure GlobalFlexAttr.

See Also

VisitAllSelectFields

2. The Selector

The first child of the AbstractSelector is the Selector. Conceptually, a Selector is much like a PickList: the user is presented with a series of choices; he moves the highlight to the one he wants; then he presses <Enter> to select it. The Selector differs from the PickList, however, in that the items can appear wherever the programmer wants them to appear. They do not have to be arranged in tidy rows and columns.

Alternatively, you might think of a Selector as a simplified version of the EntryScreen. Whereas the EntryScreen builds a list of fields that know their contents and know how to "edit themselves," in the best object oriented fashion, the Selector builds a list of fields that know only their locations and dimensions. User-written routines must be called to obtain their contents and to perform actions (such as editing) on a field once it has been selected. Nevertheless, the two objects are so closely related that it is possible to make a Selector behave very much like an EntryScreen, as we'll demonstrate in the example program shown below:

```
{.$DEFINE AutoSelect} {if defined, selector simulates entry screen}

{$V-,F+}
program SelectorExample; {EXSEL1.PAS}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEdit;
var
  Sel      : Selector;
  Finished : Boolean;
  Data     : array[0..5] of string[12];
  I        : Word;

  procedure Action(SP : SelectorPtr; ID : Word);
  var
    LE : LineEditor;
  begin
    LE.Init(DefaultColorSet);
    LE.leEditOptionsOff(leHouseCursorAtEnd);

    {$IFDEF AutoSelect}
    LE.leEditOptionsOn(leAutoAdvance);
    LE.leSecEditOptionsOff(sleNoFieldMovement);
    {$ENDIF}

    LE.ReadString('', WhereYabs, WhereXabs, 12, 12 , Data[ID]);

    {$IFDEF AutoSelect}
    if LE.GetLastCommand = ccSelect then
      SP^.SetLastCommand(ccNextField)
    else
      SP^.SetLastCommand(LE.GetLastCommand);
    {$ENDIF}

    LE.Done;
  end;

  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
  begin
    if NeedPrompt then
      S := 'Field ' + Long2Str(ID)
    else
      S := Pad(Data[ID], 12);
  end;
```

```
  procedure PreSelect(SP : SelectorPtr; ID : Word);
  begin
    FastWrite('Current Field:  ' + Long2Str(ID), 25, 1, TextAttr);
  end;

  procedure PostSelect(SP : SelectorPtr; ID : Word);
  begin
    FastWrite('Previous Field: ' + Long2Str(ID), 24, 1, TextAttr);
  end;

begin
  {instantiate Sel}
  if not Sel.InitCustom(27, 6, 52, 15,              {Window coordinates}
                        DefaultColorSet,            {ColorSet}
                        DefWindowOptions+wBordered) {Window options}
  then begin
    WriteLn('Failed to init Selector. Status = ', InitStatus);
    Halt;
  end;

  {add some window frills}
  Sel.EnableExplosions(10);
  Sel.wFrame.AddShadow(shBR, shOverWrite);
  Sel.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  Sel.wFrame.AddHeader(' Selector ', heTC);

  Sel.SetGetFieldProc(GetField);
  Sel.SetActionProc(Action);
  Sel.SetPreSelectProc(PreSelect);
  Sel.SetPostSelectProc(PostSelect);

  {$IFDEF AutoSelect}
  Sel.slOptionsOn(slAutoSelect);
  {$ENDIF}

  {add text fields}
  Sel.AddTextField('-Fixed text field-', 1, 4);
  Sel.AddTextField('-Fixed text field-', 10, 4);

  {add regular fields}
  FillChar(Data, SizeOf(Data), 0);
  for I := 1 to 6 do
    Sel.AddField(I+2, 03, 07,     {Prompt Row, Col, Width}
                 I+2, 12, 12,     {Field Row, Col, Width}
                 0);              {Help Index}

  Finished := False;
  repeat
    Sel.Process;
    case Sel.GetLastCommand of
      ccQuit :  Finished := True;
      ccError : begin
                  WriteLn('Fatal error ', Sel.GetLastError);
                  Finished := True;
                end;
      {... process other exit commands ...}
    end;
  until Finished;

  Sel.Erase;
  Sel.Done;
end.
```

Unlike most of our introductory example programs, this one is designed to illustrate the major features of a Selector, rather than its basic use. There are two reasons for the shift in emphasis here. First, none of the full-fledged demonstration programs illustrates the use of a Selector. (DPINST.PAS does, however, show how to use a ScrollingSelector.) And second, it's important to show you how the various hooks provided by the Selector work together, and a simpler example wouldn't do the job. Before we get started looking at it, we should point out that the conditional compilation directive AutoSelect will be discussed only when we're finished with other topics. Please ignore it for the time being.

Now, let's get down to business. The first task is to instantiate the Selector by calling InitCustom, which takes no special parameters, just the usual ones for all window-based objects: window coordinates, color set, and window options. In this case we've asked for a bordered window. The next few method calls add some "frills": explosions, shadows, a scroll bar and a header at the top.

The next group of calls set up some user-written routines that will be called by the Selector in various situations. These hooks will be explained in detail later on, so we'll just give you a brief idea of what they do. SetGetFieldProc specifies the routine that is to be called when the Selector needs to display the contents of a field. SetActionProc specifies the routine that is to be called when the user selects a field. SetPreSelectProc specifies a routine that is to be called when the highlight is first moved to a given field, and SetPostSelectProc specifies a routine to be called when it is about to be moved away from it. Use of these last two hooks is completely optional. A Selector cannot do anything without a GetField routine, on the other hand, and it is largely worthless if there is no Action routine.

The calls to AddTextField add some display-only fields, fields whose "values" don't change. The calls to AddField, by contrast, designate the regions of the window in which the contents of the Data array will be displayed.

Once the fields have been added, we enter a loop in which the Selector is displayed and commands are processed until the user gives an exit command. When it is first displayed, the first field (0) will be highlighted. The user can then move the cursor to another field using a cursor key, or he can select the field by pressing <Enter>. When a field is selected, the Action routine will be called. In this particular case, the Action routine is fairly simple. All the fields are strings that are to be edited, and the ID parameter can be used as an index into Data. In other cases, the Action routine might have to do very different things depending on the value of ID. When the Action routine is finished, the Selector will resume control and continue processing commands until the user presses <Esc>, at which time the Selector is erased and destroyed.

And now for the big question: What happens if AutoSelect is defined? First, the slAutoSelect option is activated. This option causes a fundamental change in the behavior of the Selector. Instead of waiting for the user to select a field, the Selector selects the current field automatically, invoking the Action routine. It then becomes the Action routine's responsibility to generate commands, because (if the Action routine is written properly) the Selector will never call its command processor–it will always assume a "command" of ccSelect.

That leads us to the other changes, in the Action routine. Because the line editor is now responsible for generating field movement commands, we activate the auto-advance option and disable the no-field movement option. Together, these two option settings insure that the cursor keys can be moved from field to field, and that the cursor will be advanced to the next field when the current field is full or the user presses <Left> or <Right> at the edge of the field. (Alternatively, we might have used 'leAutoAdvanceCursor' instead of 'leAutoAdvance'.

See the documentation for OPEDIT and OPFIELD for a discussion of the various editing options.)

And, after the call to ReadString, we also add calls to the SetLastCommand method to tell the Selector where to move the cursor next. Since ccSelect means something different to a Selector than it does to an EntryScreen, which is what we're emulating here, we have to translate it into the ccNextField command. This tells the Selector to move the cursor to the next field.

Would you ever want to use a Selector in this way? Probably not. When compiled with AutoSelect defined, this program is only marginally smaller than an equivalent program written with an EntryScreen, if at all. The idea here is simply to give you an example of a program that uses the slAutoSelect option, and to illustrate how use of that option can change the demands made of an Action routine.

Fields and Field Options

There is really very little that you need to know about the fields in a Selector. If you understood the example above, and you've read the discussion of "Protected, Hidden, and Invisible Fields" in the previous section, you already know most of it. There is one special field option that we haven't touched on yet, sfIncDec, but that one will be discussed in the section on "Action Routines," below. Beyond that, the only other discussion of relevance here is the section on the SelectField in the OPFIELD portion of this chapter, which you should read if you haven't done so already.

GetField Routines

Because the fields in a Selector keep no record of the contents of the field or even the field's prompt, the Selector must rely on a user-written routine to provide it this information when it needs to draw the field. Such a procedure is called a GetField routine, and it has the following form:

```
{$F+}
procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
begin
  if NeedPrompt then
    {return prompt for field corresponding to ID in S}
  else
    {return contents of field corresponding to ID in S} ;
end;
{$F-}
```

For any given field, GetField will be called twice: once to get the prompt for the field (NeedPrompt = True), and once to get the field's contents (NeedPrompt = False). ID indicates which field the Selector needs information about, and S is the string variable in which GetField should return that information. GetField routines are installed by calling SetGetFieldProc.

Important point: It is the responsibility of the GetField routine to return a string padded to the proper width. The caller will truncate the string passed back to it if it is too long, and it will skip the call to the GetField routine entirely if knows the result will be an empty string, but the caller will not pad a string that is too short.

Action Routines

A Selector also depends on a user-written routine to perform whatever task is appropriate for a given field once it has been selected. The most common task involves editing a value associated with the field, but it might mean something very different, such as displaying a pick list, exiting

from the program, or displaying a help screen. Whatever that task might be, it is the job of an "action routine" to perform it.

An action routine has the following form:

```
{$F+}
procedure Action(SP : SelectorPtr; ID : Word);
begin
end;
```

The SP parameter is a pointer to the selector making the call, and ID identifies the current field.

In most cases, the ID alone is enough to determine what action should be taken. But if the field is an "incrementable field"–that is, if the sfIncDec option is set for it–then SP^.GetLastCommand must be called to determine what action to take. For example, let's assume that field 0 is an incrementable field associated with an integer variable, MyInt. The action routine should contain code similar to the following:

```
case ID of
  0 : case SP^.GetLastCommand of
        ccIncChoice : Inc(MyInt);
        ccDecChoice : Dec(MyInt);
        else          {edit MyInt} ;
      end;
  {... other valid IDs ...}
end;
...
Sel.slFieldOptionsOn(sfIncDec);
Sel.AddField(...);                {<- field 0}
Sel.slFieldOptionsOff(sfIncDec);
```

Note that it is not necessary to account for the ccIncChoice and ccDecChoice commands for fields that do not have the sfIncDec option set. The Selector will ignore those commands if the option is not set for the current field.

Pre-Select Routines

A pre-select routine is a user-written procedure that is called by Process just before it moves the cursor to a new field. The most common use for this hook is to display information about the field being moved to, but it can also be used to customize the behavior of the selector. For example, it can force the selector to execute a particular command by calling SetLastCommand, or it can force the cursor to move to a different field by calling SetNextField.

A pre-select routine has the following form:

```
{$F+}
procedure PreSelect(SP : SelectorPtr; ID : Word);
begin
end;
```

The SP parameter is a pointer to the selector making the call, and ID identifies the current field.

Post-Select Routines

The counterpart of the pre-select routine is the post-select routine, a user-written procedure that is called by Process when it is about to exit or to move the cursor to another field. Like the pre-select routine, the post-select routine can be used to customize the behavior of the selector.

For example, it can force the selector to execute a particular command by calling SetLastCommand, or it can force the cursor to move to a different field by calling SetNextField.

A post-select routine has the following form:

```
{$F+}
procedure PostSelect(SP : SelectorPtr; ID : Word);
begin
end;
```

The SP parameter is a pointer to the selector making the call, and ID identifies the current field.

Selection Commands

The commands offered by OPSELECT for Selectors and ScrollingSelectors are arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant SelectKeySet.

Field Movement

ccPrevField **<ShiftTab>**
Move cursor to previous unprotected field.

ccNextField **<Tab>, <CtrlI>**
Move cursor to next unprotected field.

ccLeft **<Left>, <CtrlS>**
Cursor left one field.

ccRight **<Right>, <CtrlD>**
Cursor right one field.

ccHome **<Home>, <CtrlQ><S>**
Cursor to first unprotected field on current row.

ccEnd **<End>, <CtrlQ><D>**
Cursor to last unprotected field on current row.

ccUp **<Up>, <CtrlE>**
Cursor up one row.

ccDown **<Down>, <CtrlX>**
Cursor down one row.

ccPageUp **<PgUp>, <CtrlR>**
Scroll display up one page.

ccPageDn **<PgDn>, <CtrlC>**
Scroll display down one page.

ccFirstField **<CtrlPgUp>, <CtrlQ><R>**
Move cursor to first unprotected field in the list.

ccLastField **<CtrlPgDn>, <CtrlQ><C>**
Move cursor to last unprotected field in the list.

Field Selection

ccSelect **<Enter>, <CtrlM>, <ClickCenter>**
Select the current field (call the Action routine).

ccMouseSel **<ClickLeft>**
Moves the cursor to the field indicated by the mouse. If that field is already highlighted, it is selected.

ccIncChoice **<+>, <Space>**
If the current field is designated as an incrementable field, this command selects the field. The Action routine can distinguish between ccIncChoice and ccSelect by calling the Selector's GetLastCommand method.

ccDecChoice **<->**
If the current field is designated as an incrementable field, this command selects the field. The Action routine can distinguish between ccDecChoice and ccSelect by calling the Selector's GetLastCommand method.

Other

ccDone **<CtrlEnter>, <CtrlJ>, <CtrlK><D>, <CtrlK><Q>**
Exit from the Selector.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling SelectCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a GetHelpProc, the Selector will pass it the following three parameters: ucSelect, @Self, and the help index for the current field.

ccQuit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit the selector.

In addition to the commands shown above and the standard exit commands (ccError and the user exit commands--ccUser0, etc.), Process also recognizes several other commands that might be passed to it by a LineEditor via an Action routine (see the example program at the start of the section). These are: ccTab and ccAutoAdvance (treated like ccNextField), ccBackTab (treated like ccPrevField), ccWordLeft (treated like ccLeft), and ccWordRight (treated like ccRight).

Constants

```
BadSelectOptions : Byte = slFastUpdates;
```

Selector options that exist for internal use, and may not be altered by calling slOptionsOn or slOptionsOff.

```
DefSelectOptions : Byte = slMousePage;
```

The default options for a Selector or ScrollingSelector.

```
SelectKeyMax = 160;
SelectKeyID  : string[13] = 'opselect keys';
SelectKeySet : array[0..SelectKeyMax] of Byte = (...);
SelectCfgEnd : Byte = 0;
```

SelectKeyId is an ID string used to mark the the beginning of the configuration data area for OPSELECT; SelectCfgEnd marks the end of the configuration data area. In between them is SelectKeySet, the command table used by SelectCommands. SelectKeyMax is the last valid index into the table.

```
slScrollbyPage    = $01; {scroll by page?}
slMousePage       = $02; {clicking on scroll bar scrolls by page}
slAutoSelect      = $20; {auto-select current field?}
slFastUpdates     = $80; {used internally}
```

Options for a Selector or ScrollingSelector. slScrollbyPage affects the vertical scrolling behavior of a ScrollingSelector when the cursor is moved to a field not currently on screen. If the option is off, the display will be scrolled only enough to reveal the field to move to. If the option is on, the display will be scrolled by a full page (usually equal to the number of rows that can be displayed at one time). slMousePage also applies only to a ScrollingSelector, and it determines what happens when you click on the increment or decrement hot spots of a scroll bar. If the option is off, clicking on these spots is equivalent to pressing <Down> or <Up>, respectively. If the option is on, clicking on them is equivalent to pressing <PgDn> or <PgUp>.

The slAutoSelect option indicates that the Process method should automatically select the field that the cursor is on, rather than waiting for the user to press <Enter> to select it. See the example program at the start of this section for an illustration of how to use this option. The slFastUpdates option is an internal flag used to speed up screen updates; it should not be modified.

Types

```
GetFieldProc = procedure(ID : Word; NeedPrompt : Boolean; var S : string);
```

A user-written procedure called to obtain a string (S) containing the prompt (NeedPrompt = True) or contents (NeedPrompt = False) of a specified field (ID).

```
SelectorPtr = ^Selector;
Selector =
  object(AbstractSelector)
    ...
  end;
```

An object that allows the user to select fields from a list.

```
SelectProc = procedure(SP : SelectorPtr; ID : Word);
```

A user-written procedure called when 1) the cursor is about to be moved to a new field (a pre-select routine), 2) a field has been selected by the user (an action routine), or 3) the cursor is about to leave the current field (a post-select routine).

Variables

```
SelectCommands : CommandProcessor;
```

The default CommandProcessor for a Selector or ScrollingSelector.

AddField

Declaration

```
procedure AddField(PromptRow, PromptCol : Word; PromptWidth : Byte;
                   FieldRow,  FieldCol : Word;  FieldWidth : Byte;
                   HelpIndex : Word);
```

Purpose

Add a field with the specified coordinates and help index.

Description

This method is used to add a field to a Selector.

PromptRow and PromptCol specify the (relative) screen coordinates for the prompt, and PromptWidth its width. FieldRow and FieldCol specify the (relative) screen coordinates for the field itself, and FieldWidth its width. Note that, if PromptWidth = 0 (meaning that the field has no prompt), PromptRow and PromptCol will be set equal to FieldRow and FieldCol, respectively.

HelpIndex is the value to be passed to the GetHelpProc associated with SelectCommands, if any, when this particular field is selected. Note that, if the request for help is made while the Action routine is in control, it will be up to it to see that the correct index is passed to the help routine. If OPEDIT is being used, for example, this can be accomplished by calling the LineEditor's SetHelpIndex method. The help index for a given field can be obtained by calling GetHelpIndex with the ID number for the field.

There are two sets of limits imposed on row and column coordinates for prompts and fields, one for Selectors and one for ScrollingSelectors. In the case of a Selector, all the prompts and fields must fit entirely within the inner portion of the Selector's window. For example, if the window is 15 rows high, then PromptRow and FieldRow must be in the range 1..15. If the window is 60 columns wide then the expressions (PromptCol+PromptWidth-1) and (FieldCol+FieldWidth-1) must both be in the range 1..60. If the coordinates are not in the proper range, an error (epFatal+ecBadCoordinates) will be generated that will cause any subsequent calls to AddField to fail as well. The error condition can be detected by calling GetLastError.

For a ScrollingSelector, there are both absolute and practical limits. The absolute limit is a function of the fact that a virtual screen must fit in a single 64K segment and must start on a segment boundary. Consequently, the product of the maximum row and maximum column (for any prompt or field in the selector) must be less than 32753 (= 65506/2). Practically speaking, however, you will probably not want to create a ScrollingSelector that approaches the absolute limit. A scrolling selector with 400 fields will generally perform acceptably on a 10 MHz '286 machine, for example, but would produce noticeable delays on an early model PC.

Example

```
Sel.AddField(2, 01, 10,     {Prompt Row, Col, Width}
             2, 12, 10,     {Field Row, Col, Width}
             0);            {Help Index}
```

Adds one selector field to Sel. The prompt will be displayed at row 2, column 1; the field will be displayed at row 2, column 12.

See Also

GetHelpIndex

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Create a selector using default window options.

Description

This constructor instantiates a Selector using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not Sel.Init(30, 5, 55, 14) then begin
  WriteLn('Failed to init Selector. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a selector using default window options and colors.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
```

Purpose

Create a selector with custom window options.

Description

This constructor instantiates a Selector using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window.

InitCustom will fail only if the parent constructor (AbstractSelector.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Several sets of attributes in the specified ColorSet have special relevance for window objects of type Selector (or ScrollingSelector). See the entry for AbstractSelector.InitCustom for details.

Example

```
if not Sel.InitCustom(30, 5, 55, 14,              {Window coordinates}
                      DefaultColorSet,            {ColorSet}
                      DefWindowOptions or wBordered) {Window options}
then begin
  WriteLn('Failed to init Selector. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a selector with a bordered window and default window colors.

See Also

AbstractSelector.InitCustom

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a selector from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Selector.Store. Note that any user-written routines associated with the Selector (Action, GetField, PreSelect, PostSelect) should have their addresses registered with the stream before calling Load, as shown in the example below.

The stream registration procedure for a Selector is SelectorStream.

Example

```
{$V-,F+}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpField, OpSelect, OpEdit;
var
  Sel : Selector;
  S : BufIdStream;
  Data : array[0..5] of String[10];
  I : Word;

  procedure Action(SP : SelectorPtr; ID : Word);
  begin
    {...}
  end;
  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
  begin
    {...}
  end;

begin
  S.Init('SELECT.STM', SOpen, 1024);
  S.RegisterHier(SelectorStream);
  S.RegisterPointer(1000, @Action);
  S.RegisterPointer(1001, @GetField);
  S.Get(Sel);
  if S.GetStatus <> 0 then
    Halt;
  S.Done;

  FillChar(Data, SizeOf(Data), 0);
  ClrScr;
  Sel.Process;
  Sel.Erase;
  Sel.Done;
end.
```

In this example, we load Sel from a stream created by the Store example using the stream's Get method, which calls Selector.Load indirectly. Note that the Store example registered pointers to its Action and GetField procedures with codes of 1000 and 1001, respectively, so we must register our procedures with the same codes.

See Also

Store

Declaration
```
procedure Process; virtual;
```

Purpose
Process selection commands.

Description
Selector's Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Selection Commands" at the start of this section.

If the field designated as the current field is protected, Process will move the highlight to the first unprotected field it can find. If there are no unprotected fields, Process exits with the error epFatal+ecNoFields.

Process makes direct calls to up to 3 user-written routines. Before moving the highlight to a new field, Process calls the pre-select routine. When the user selects a field, it calls the Action routine. And before leaving the current field it calls the post-select routine. See the discussion of these hooks at the beginning of this section for details.

Example
See example at the beginning of this section.

See Also
CommandWindow.Process SetStatusProc

Declaration
```
procedure SetActionProc(AP : SelectProc);
```

Purpose
Set routine to call when a field is selected.

Description
For each selector, this method must be called once to specify the action routine (AP) that is to be called when a field has been selected.

An action routine must be of the following form:

```
{$F+}
procedure Action(SP : SelectorPtr; ID : Word);
begin
  {perform appropriate action for field specified by ID}
end;
```

The SP parameter is a pointer to the selector making the call (@Self is passed by Process). ID indicates the field that was selected. See the discussion of action routines at the beginning of this section for further details.

Example
See the example program at the beginning of this section.

See Also
SetGetFieldProc

SetGetFieldProc / SetPostSelectProc

Declaration

```
procedure SetGetFieldProc(GFP : GetFieldProc);
```

Purpose

Set routine to call to get strings to display.

Description

For each selector, this method must be called once to specify the GetField routine (GFP) that is to be called when the selector needs to draw the prompt for or contents of a particular field.

A GetField routine must be of the following form:

```
{$F+}
procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
begin
  if NeedPrompt then
    {return prompt for field corresponding to ID in S}
  else
    {return contents of field corresponding to ID in S} ;
end;
```

ID indicates the field to be drawn. NeedPrompt will be True if the prompt is needed, and False if the contents of the field are needed. S is the string to be returned. See the discussion of GetField routines at the beginning of this section for further details.

Example

See the example program at the beginning of this section.

See Also

SetActionProc

Declaration

```
procedure SetPostSelectProc(PSP : SelectProc);
```

Purpose

Set routine to call before moving to next field.

Description

This method allows you to specify a post-select routine (PSP), which will be called by Process just before the cursor is moved from the current field.

A post-select routine must be of the following form:

```
{$F+}
procedure PostSelect(SP : SelectorPtr; ID : Word);
begin
end;
```

The SP parameter is a pointer to the selector making the call (@Self is passed by Process). ID identifies the current field. See the discussion of post-select routines at the beginning of this section for further details.

Example

See the example program at the beginning of this section.

See Also

SetPreSelectProc

Declaration
```
procedure SetPreSelectProc(PSP : SelectProc);
```

Purpose
Set routine to call before moving to new field.

Description
This method allows you to specify a pre-select routine (PSP), which will be called by Process just before the cursor is moved to a new field.

A pre-select routine must be of the following form:

```
{$F+}
procedure PreSelect(SP : SelectorPtr; ID : Word);
begin
end;
```

The SP parameter is a pointer to the selector making the call (@Self is passed by Process). ID identifies the current field. See the discussion of pre-select routines at the beginning of this section for further details.

Example
See the example program at the beginning of this section.

See Also
SetPostSelectProc

Declaration
```
function slFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified field options are on.

Description
This method returns True if the specified field option(s) are currently selected.

Example
```
if Sel.slFieldOptionsAreOn(sfMapCtrls) then
  Sel.slFieldOptionsOff(sfMapCtrls)
else
  Sel.slFieldOptionsOn(sfMapCtrls);
```

Toggle the state of the sfMapCtrls option.

See Also
slFieldOptionsOff slFieldOptionsOn

slFieldOptionsOff / slFieldOptionsOn

Declaration
```
procedure slFieldOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified field option(s), excluding those designated in OPFIELD as BadSFieldOptions (see the documentation for OPFIELD).

Example
```
Sel.slFieldOptionsOff(sfProtected);
```
Turns off the protected field option.

See Also
slFieldOptionsAreOn

Declaration
```
procedure slFieldOptionsOn(OptionFlags : LongInt);
```

Purpose
Activate multiple options.

Description
This method activates the specified field option(s), excluding those designated in OPFIELD as BadSFieldOptions (see the documentation for OPFIELD).

Example
```
Sel.slFieldOptionsOn(sfProtected);
Sel.AddField(2, 01, 10,      {Prompt Row, Col, Width}
             2, 12, 10,      {Field Row, Col, Width}
             0);             {Help Index}
Sel.slFieldOptionsOff(sfProtected);
```
Add a protected field.

See Also
slFieldOptionsAreOn

Declaration
```
function slOptionsAreOn(OptionFlags : Byte) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if the specified Selector option(s) are currently selected.

Example
```
if Sel.slOptionsAreOn(slScrollByPage) then
  Sel.slOptionsOff(slScrollByPage)
else
  Sel.slOptionsOn(slScrollByPage);
```
Toggles option to scroll by page.

Declaration
```
procedure slOptionsOff(OptionFlags : Byte);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified Selector option(s), excluding those designated as BadSelectOptions (see the Constants section, above).

Example
```
Sel.slOptionsOff(slScrollByPage);
```
Turns off option to scroll by page.

Declaration
```
procedure slOptionsOn(OptionFlags : Byte);
```

Purpose
Activate multiple options.

Description
This method activates the specified Selector option(s), excluding those designated as BadSelectOptions (see the Constants section, above).

Example
```
Sel.slOptionsOn(slAutoSelect);
```
Activates auto-select option.

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a selector in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Selector to the stream S. Note that any user-written routines associated with the Selector (Action, GetField, PreSelect, PostSelect) should have their addresses registered with the stream before calling Store, as shown in the example below.

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a Selector is SelectorStream.

Example

```
{$V-,F+}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpField, OpSelect, OpEdit;
var Sel : Selector;
    S : BufIdStream;
    Data : array[0..5] of String[10];
    I, Status : Word;

  procedure Action(SP : SelectorPtr; ID : Word);
  begin
    ...
  end;
  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
  begin
    ...
  end;

begin
  FillChar(Data, SizeOf(Data), 0);
  if not Sel.InitCustom(30, 5, 55, 14, DefaultColorSet,
                        DefWindowOptions or wBordered) then
    Halt;

  Sel.SetGetFieldProc(GetField);
  Sel.SetActionProc(Action);
  for I := 1 to 6 do
    Sel.AddField(I+2, 1, 10, I+2, 12, 10, 0);

  S.Init('SELECT.STM', SCreate , 1024);
  S.RegisterHier(SelectorStream);
  S.RegisterPointer(1000, @Action);
  S.RegisterPointer(1001, @GetField);
  S.Put(Sel);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status); Halt;
  end;
  S.Done;
  Sel.Done;
end.
```

Instantiates Sel, adds fields to it, and then stores Sel in a stream using the stream's Put method, which calls Store indirectly. Note that we've registered pointers to the Action and GetField procedures with codes of 1000 and 1001, respectively. See Load for an example program that reloads this Selector.

See Also

Load

3. The ScrollingSelector

The Selector object described in the previous section has two important limitations: 1) all the fields associated with it must fit within the window at one time, and 2) the window cannot be zoomed or resized. The purpose of the ScrollingSelector is to provide a Selector-based object that lacks these limitations.

Compare the following program to the one at the beginning of the previous section:

```
{$V-,F+}
program ScrollingSelectorExample; {EXSEL2.PAS}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEdit;
var
  SSel     : ScrollingSelector;
  Finished : Boolean;
  Data     : array[0..29] of string[12];
  I        : Word;

  procedure Action(SP : SelectorPtr; ID : Word);
    {-Called by Selector when Enter is pressed}
  var
    LE : LineEditor;
  begin
    LE.Init(DefaultColorSet);
    LE.leEditOptionsOff(leHouseCursorAtEnd);
    LE.ReadString('', WhereYabs, WhereXabs, 12, 12 , Data[ID]);
    LE.Done;
  end;

  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
    {-Called by Selector to display field strings}
  begin
    if NeedPrompt then
      S := 'Field '+LeftPad(Long2Str(ID), 2)
    else
      S := Pad(Data[ID], 12);
  end;

begin
  FillChar(Data, SizeOf(Data), 0);

  if not SSel.InitCustom(27, 6, 52, 15,                {Window coordinates}
                         DefaultColorSet,              {ColorSet}
                         DefWindowOptions OR wBordered) {Window options}
  then begin
    WriteLn('Failed to init Selector. Status = ', InitStatus); Halt;
  end;

  {add some window frills}
  SSel.EnableExplosions(10);
  SSel.wFrame.AddShadow(shBR, shOverWrite);
  SSel.wFrame.AddScrollBar(frRR, 0, MaxLongInt, DefaultColorSet);
  SSel.wFrame.AddHeader(' Scrolling Selector ', heTC);

  {selector options}
  SSel.SetWrapMode(StopAtEdges); {<-- best setting for a ScrollingSelector}

  {set procedure pointers}
  SSel.SetGetFieldProc(GetField);
  SSel.SetActionProc(Action);
```

```
    {add fields}
    for I := 1 to 30 do
      SSel.AddField(I, 03, 08,      {Prompt Row, Col, Width}
                    I, 12, 12,      {Field Row, Col, Width}
                    0);             {Help Index}
    SSel.AddTextField('-Fixed text field-', 31, 4);

    {allocate the virtual screen}
    SSel.AllocateScreen;

    {check for errors}
    I := SSel.GetLastError;
    if I <> 0 then begin
      WriteLn('Error ', I, ' creating selector');
      Halt;
    end;

    Finished := False;
    repeat
      SSel.Process;
      case SSel.GetLastCommand of
        ccQuit :  Finished := True;
        ccError :
          begin
            WriteLn('Fatal error ', SSel.GetLastError);
            Finished := True;
          end;
        {... process other exit commands ...}
      end;
    until Finished;

    {erase and dispose of selector}
    SSel.Erase;
    SSel.Done;
  end.
```

Apart from the absence of pre-select and post-select routines, the only differences between this program and the earlier one are 1) this one has more fields, 30 instead of 6, 2) this one has a call to AllocateScreen after all of the fields have been added, and 3) GetLastError is called after AllocateScreen. The most important of these is the second.

What AllocateScreen does is to allocate a virtual screen on which all of the fields in the list can be drawn at once. Once the entire virtual screen has been filled in, the ScrollingSelector will determine which section of the virtual screen needs to be copied to the physical screen in order to display the current field, then copy it. As the cursor is moved from field to field, the ScrollingSelector will check to see whether or not the new field is already being displayed on the screen. If it is, only the one field needs to be redrawn. If not, it will quickly copy a different section of the (already drawn) virtual screen to the physical screen, then redraw the current field. Although the same effect could be achieved without use of a virtual screen, screen updates would be significantly slower.

Because virtual screens can be very large–nearly 64K–it is possible to have literally hundreds of fields associated with a single ScrollingSelector (400 is a good upper limit). In the case of our example program, we have 30 fields displayed within a window that is 10 rows high. If you run the program and press <Down> repeatedly, you'll see the first 10 fields progressively scroll out of view once you reach the bottom of the window, and the last 20 fields will scroll into view.

The ScrollingSelector depends so heavily on its virtual screen that it cannot even draw itself once without it. This is the reason for the call to GetLastError in the example program. If AllocateScreen failed to allocate the virtual screen, there's no point in going any farther. Although it will be called for you automatically if you fail to do it, the error condition will generally be easier to handle if you make the call yourself.

Even if all your fields fit into a Selector's window, you may want to use a ScrollingSelector anyway, because it can be zoomed and resized. This is a happy by-product of the technique that ScrollingSelector uses to update the window. Since it cannot assume, as Selector must, that all the fields will fit in the window, the size of the window doesn't really matter as long as the largest field in the list will fit. (ScrollingSelector ensures that it will by setting a minimum size for the window.)

For an example of a real program that uses a ScrollingSelector, see DPINST.PAS.

Important: Please be sure to read the discussion of the ResetScreen method in the introduction to the AbstractSelector section before using a ScrollingSelector.

Types

```
ScrollingSelector =
  object(Selector)
    ...
    ssRowOfs     : Integer;
    ssColOfs     : Integer;
    ...
  end;
```

A scrollable version of the Selector object. The only data fields of general interest in a ScrollingSelector are ssRowOfs and ssColOfs. ssRowOfs keeps track of the number of rows that have been scrolled out of view: if row 8 was being displayed at the top of the window, for example, ssRowOfs would be 7. ssColOfs is the number of columns that have been scrolled out of view: if column 25 was being displayed at the left edge of the window, ssColOfs would be 24.

Declaration
```
procedure AllocateScreen;
```

Purpose
Call after last field added to allocate virtual screen.

Description
This method should be called after the last field has been added to the ScrollingSelector. It scans the list of fields (including text fields) to determine the maximum row and column values for any field in the list, then tries to allocate a virtual screen large enough to allow all the fields to be drawn on it at once. To determine whether or not the virtual screen was allocated successfully, you should call GetLastError immediately following the call to AllocateScreen. Assuming that there was no error pending prior to the call, the only error code GetLastError should return is epFatal+ecOutOfMemory.

Note: If you neglect to call AllocateScreen before calling Draw or Process, AllocateScreen will be called for you. In most cases, however, it is easier to handle errors if you call AllocateScreen manually.

Example
```
SSel.AllocateScreen;
if SSel.GetLastError <> 0 then begin
  WriteLn('Failed to allocate screen');
  Halt;
end;
```

After adding all selector fields and text fields, we call AllocateScreen to allocate an appropriate sized virtual screen.

See Also
CommandWindow.GetLastError Selector.AddField

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate field list, screen buffers.

Description
This destructor disposes of the virtual screen associated with the ScrollingSelector, then calls the parent's destructor (AbstractSelector.Done) to dispose of the field list, screen buffers, etc.

Example
```
SSel.Erase;
SSel.Done;
```

Erase SSel and release all heap space.

See Also
AbstractSelector.Done

Init / InitCustom

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Create a scrollable selector using default window options.

Description

This constructor instantiates a ScrollingSelector using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not SSel.Init(30, 5, 55, 14) then begin
  WriteLn('Failed to init Selector. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a scrollable selector using default window options and colors.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
```

Purpose

Create a scrollable selector with custom window options.

Description

This constructor instantiates a Selector using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window.

InitCustom will fail only if the parent constructor (Selector.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Several sets of attributes in the specified ColorSet have special relevance for window objects of type ScrollingSelector. See the entry for AbstractSelector.InitCustom for details.

Example

```
if not SSel.InitCustom(30, 5, 55, 14,                {Window coordinates}
                       DefaultColorSet,              {ColorSet}
                       DefWindowOptions or wBordered) {Window options}
then begin
  WriteLn('Failed to init Selector. Status = ', InitStatus);
  Halt;
end;
```

Instantiates a scrollable selector with a bordered window and default window colors.

See Also

AbstractSelector.InitCustom Selector.InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a selector from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to ScrollingSelector.Store. Note that any user-written routines associated with the ScrollingSelector (Action, GetField, PreSelect, PostSelect) should have their addresses registered with the stream before calling Load, as shown in the example below.

Note too that it is not necessary to call AllocateScreen after a call to Load. Load does that for you.

The stream registration procedure for a ScrollingSelector is ScrollingSelectorStream.

Example

```
{$V-,F+}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEdit;
var
  SSel : ScrollingSelector;
  S    : BufIdStream;
  Data : array[0..29] of string[10];
  I    : Word;
  procedure Action(SP : SelectorPtr; ID : Word);
  begin
    {...}
  end;
  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
  begin
    {...}
  end;
begin
  {Load SSel from a stream}
  S.Init('SELECT.STM', SOpen , 1024);
  S.RegisterHier(ScrollingSelectorStream);
  S.RegisterPointer(1000, @Action);
  S.RegisterPointer(1001, @GetField);
  S.Get(SSel);
  if S.GetStatus <> 0 then Halt;
  S.Done;
  FillChar(Data, SizeOf(Data), 0);
  SSel.Process;
  SSel.Done;
end.
```

In this example, we load SSel from a stream created by the Store example using the stream's Get method, which calls ScrollingSelector.Load indirectly. Note that the Store example registered pointers to its Action and GetField procedures with codes of 1000 and 1001, respectively, so we must register our procedures with the same codes.

See Also

Selector.Load Store

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a ScrollingSelector in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the ScrollingSelector to the stream S. Note that any user-written routines associated with the object (Action, GetField, PreSelect, PostSelect) should have their addresses registered with the stream before calling Store, as shown below.

To check for errors that may have occurred during the Store, call the stream's GetStatus function, as shown below.

The stream registration procedure for a ScrollingSelector is ScrollingSelectorStream.

Example

```
{$V-,F+}
uses
  OpCrt, OpString, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEdit;
var SSel : ScrollingSelector;
    S : BufIdStream;
    Data : array[0..29] of string[10];
    I, Status : Word;

  procedure Action(SP : SelectorPtr; ID : Word);
  begin
    {...}
  end;
  procedure GetField(ID : Word; NeedPrompt : Boolean; var S : string);
  begin
    {...}
  end;

begin
  FillChar(Data, SizeOf(Data), 0);
  if not SSel.InitCustom(30, 5, 55, 14, DefaultColorSet,
                         DefWindowOptions+wBordered) then
    Halt;

  SSel.SetGetFieldProc(GetField);
  SSel.SetActionProc(Action);
  for I := 1 to 30 do
    SSel.AddField(I, 1, 10, I, 12, 10, 0);

  S.Init('SELECT.STM', SCreate , 1024);
  S.RegisterHier(ScrollingSelectorStream);
  S.RegisterPointer(1000, @Action);
  S.RegisterPointer(1001, @GetField);
  S.Put(SSel);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status); Halt;
  end;
  S.Done;
  SSel.Done;
end.
```

Instantiates SSel, adds fields to it, and then stores SSel in a stream using the stream's Put method, which calls Store indirectly. Note that we've registered pointers to the Action and GetField procedures with codes of 1000 and 1001, respectively. Note too that it is not necessary to call AllocateScreen in this particular case because SSel is destroyed immediately after it is stored. See Load for an example program that reloads this ScrollingSelector.

See Also

Load

Selector.Store

VScreenToScreen

Declaration

```
procedure VScreenToScreen; virtual;
```

Purpose

Copy the virtual screen to the physical screen.

Description

This method is intended strictly for internal use, and should never be called directly. It is documented here only because it is one that you might want to override in an object derived from a ScrollingSelector. What it does is copy the relevant portion of the virtual screen to the physical screen when the selector is being redrawn in its entirety or scrolled. But it is not the function of the method itself but rather the time at which it is called that makes it useful in derived types.

Suppose, for example, that you wanted a selector that scrolls horizontally to have a row of column labels displayed just above the first row of fields, and that these column labels needed to be displayed at all times, even if the selector has been scrolled vertically. (This rules out the use of text fields or a screen update procedure, since these hooks are used to display static text within the scrollable portion of the window.) Furthermore, the column labels need to change in unison with the contents of the selector as it is scrolled horizontally. (If you can't picture this situation, try popping up the quick reference chart in DESKPOP, then scrolling the window in all directions using the cursor keys. The row just below the border, which scrolls horizontally but not vertically, contains column labels.) The best way to accomplish this is to override the VScreenToScreen method, as shown in the sample code below.

Example

```
type
  MySelector =
    object(ScrollingSelector)
      constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                             Options : LongInt);
      procedure VScreenToScreen; virtual;
    end;

constructor MySelector.InitCustom(X1, Y1, X2, Y2 : Byte;
                                  var Colors : ColorSet; Options : LongInt);
begin
  if not ScrollingSelector.InitCustom(X1, Y1+2, X2, Y2, Colors,
                                      Options or wBordered) then
    Fail;
  {adjust frame}
  with wFrame do
    AdjustFrameCoords(frXL, frYL-2, frXH, frYH);
  if cwGetLastError = 0 then
    {add window divider}
    wFrame.AddSpanHeader(#195, #196, #180, 2, frTT);
  if cwGetLastError <> 0 then begin
    InitStatus := cwGetLastError;
    Done;
    Fail;
  end;
end;
procedure MySelector.VScreenToScreen;
var S : string;
begin
  ScrollingSelector.VScreenToScreen;
  {...construct a full row of column labels in S...}
  fFastWrite(Copy(S, ssColOfs+1, Width), 1, 1, TextAttr);
end;
```

MySelector is a specialized object type derived from ScrollingSelector. The InitCustom constructor serves four purposes: 1) it insures that the VMT pointer for the object is set up correctly; 2) it insures that the window has a border; 3) it adjusts the coordinates for the inner portion of the window to set aside two rows above it that won't be scrolled; and 4) it adds a span header at the second row, leaving one row at the top for column labels.

VScreenToScreen does three things: 1) it calls the parent's VScreenToScreen method to update the scrollable portion of the window; 2) it constructs a full-sized row of column labels (i.e., one as wide as the virtual screen) and stores it in S; and 3) it displays the relevant portion of S on the screen row between the border and the span header.

Additional examples of this basic technique may be found in the OPCAL (Calendar object) and OPCALC (Calculator object) units. Although the objects in these units accomplish their tasks by overriding the UpdateContents method, the principle is the same.

See Also

SetScreenUpdateProc

F. OPENTRY: Data Entry Screens

In the previous section, we looked at the first half of the hierarchy based on the AbstractSelector. In this section, we'll look at the other half:

```
[CommandWindow]
  [AbstractSelector]
    [Selector]
      [ScrollingSelector]
    EntryScreen
      ScrollingEntryScreen
```

The AbstractSelector object introduced the concept of lists of fields to be displayed on screen as a group and, at least in theory, the concept of cursor movement from field to field. The Selector put these concepts into action, but with some important limitations. It doesn't know much about the fields it is displaying other than their locations and dimensions. And in order to do anything more than move a highlight bar from one field to another it requires the assistance of a user-written routine.

The EntryScreen, on the other hand, is smarter. It knows a lot more about the fields in its list. It can automatically invoke the appropriate routines to convert raw data into displayable strings, to edit those strings, to validate their contents, and to convert them back again.

You might also think of the difference between the two objects in terms of the following "equations":

```
AbstractSelector + list of SelectFields = Selector
AbstractSelector + list of EntryFields = EntryScreen
```

There are other differences between them, of course, but these equations should at least give you an idea of what their basic building blocks are.

The ScrollingEntryScreen is to the EntryScreen what the ScrollingSelector is to the Selector. That is, it differs from the EntryScreen in the same two respects: it allows you to define a list of fields that cannot be displayed all at once within the entry screen's window, and it allows that window to be zoomed and resized. A non-scrolling EntryScreen is constrained by the limits of its window.

1. The EntryScreen

The data entry system implemented in OPENTRY is as complex as it is powerful, and you shouldn't expect or even try to master it in an evening. The best approach is to concentrate on its most basic methods and features at first, get a general sense of how the individual pieces fit together, and ignore the more esoteric features until you start to feel comfortable with it.

To help you get started, let's look at a simple example program that demonstrates some of the basic features of the EntryScreen:

```
{$V-}                          {<- required for OPENTRY}
program EntryExample; {EXENTRY1.PAS}
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpField, OpSelect, OpEntry;
type
  InfoRec = record
              Name : string[30];
              Age : Byte;
              Wage, Weekly : Real;
            end;
```

```
var
  Info : InfoRec;
  ES : EntryScreen;
  Finished : Boolean;

  {$F+}
  procedure PreEdit(ESP : EntryScreenPtr);
  var
    S : string[80];
  begin
    case ESP^.GetCurrentID of
      0 : S := 'Last name, first name';     {Info.Name}
      1 : S := 'Enter age [1-150]    ';     {Info.Age}
      2 : S := 'Enter hourly wage    ';     {Info.Wage}
    end;
    ESP^.wFastWrite(S, 7, 4, TextAttr);
  end;

  procedure PostEdit(ESP : EntryScreenPtr);
  begin
    case ESP^.GetCurrentID of
      0, 1 : {no post-edit activity required};
      2 : begin
            Info.Weekly := 40.0 * Info.Wage;
            ESP^.DrawField(3);
          end;
      3 : {protected field};
    end;
  end;
  {$F-}

begin
  ClrScr;
  FillChar(Info, SizeOf(Info), 0);
  if not ES.InitCustom(25, 5, 55, 11,                {Window coordinates}
                       DefaultColorSet,              {ColorSet}
                       DefWindowOptions OR wBordered) {Window options}
  then begin
    WriteLn('Failed to init EntryScreen. Status = ', InitStatus);
    Halt;
  end;

  {add some window frills}
  ES.wFrame.AddShadow(shBR, shOverWrite);
  ES.wFrame.AddHeader(' Entry Screen ', heTC);

  {add fields}
  ES.esFieldOptionsOn(efClearFirstChar);
  ES.AddStringField('Name:', 2,1, '',        2,9, 20, 0,          Info.Name);
  ES.AddByteField('Age:',    3,1, '999',     3,9,     0, 1, 150,  Info.Age);
  ES.AddRealField('Wage:',   4,1, '99.99',   4,9,     0, 0, 0, 0, Info.Wage);
  ES.esFieldOptionsOn(efProtected);
  ES.AddRealField('Weekly:', 5,1, '9999.99', 5,9,     0, 0, 0, 0, Info.Weekly);
  ES.esFieldOptionsOff(efProtected);

  {set procedure pointers}
  ES.SetPreEditProc(PreEdit);
  ES.SetPostEditProc(PostEdit);

  Finished := False;
```

```
    repeat
      ES.Process;
      case ES.GetLastCommand of
        ccDone, ccQuit : Finished := True;
        ccError :
          begin
            WriteLn('Fatal error ', ES.GetLastError);
            Finished := True;
          end;
        {... process other exit commands ...}
      end;
    until Finished;

    ES.Erase;
    ES.Done;
  end.
```

The first thing to notice here is the compiler directive {$V-} at the start of the program. This directive is required for almost every program or unit written with OPENTRY. If you omit it, you can expect to see a compiler error when you try to add a field of type string to an entry screen.

In the USES list, you'll notice that the first few units are pretty much typical of any program that uses an object derived from the CommandWindow. The last three are not. OPFIELD is needed because it defines numerous types and constants relating to the fields in an EntryScreen. OPABSFLD and OPFEDIT do as well, but using them is generally optional. And OPSELECT is needed because it defines the AbstractSelector, from which the EntryScreen is derived.

Our example program is typical of most OPENTRY applications in that each of the fields in the EntryScreen corresponds to a field in a record variable—in this case Info, which is of type InfoRec. This is not a hard-and-fast requirement, though. A field in an entry screen can correspond to any variable. Our examples use record variables only because most real-world applications do.

Let's skip ahead to the main block of the program. After the screen is cleared, the first order of business is to initialize the data to be edited (Info). This step is important because the EntryScreen never initializes a field's value for you. If the variable associated with a field contains garbage, you'll get garbage on the screen.

Now we're ready to initialize the EntryScreen object (ES) itself. The constructors for the EntryScreen are the same as those for most other objects derived from a CommandWindow. In the case of InitCustom, the first four parameters are the coordinates for the inner window, the next is the ColorSet to use, and the last are the window options. The only thing special here is that we've activated the wBordered option. After the EntryScreen has been instantiated, though, we add a few "frills": a see-through shadow and a centered header at the top of the window border.

The next big step is to add fields to the EntryScreen. Before doing so, we activate the efClearFirstChar field option, because we want it to apply to all of the fields in the entry screen. (See the OPFIELD section for an explanation of what it does.) The first field added is a string field, the most common type. The first three parameters to AddStringField concern the field's prompt, 'Name:', which is to be displayed at row 2, column 1. The next parameter is the picture mask for the field. AddStringField is a bit unusual in that it will supply a default picture mask if none is specified. In this case, it will supply a mask 20 characters long consisting of 'X'

characters, which means that the user may enter any character in the field. The value 20 comes from the field's width, which is specified in the parameter just following the coordinates for the field itself (2,9). The next-to-last parameter, the 0, is a help index, which we don't need in this application. And the final parameter is the name of the variable associated with the field.

Most of the AddField methods take parameters similar to those required for AddStringField. These methods are all discussed in the next section, so we won't go over the calls to AddByteField and AddRealField in detail here. We will point out, however, that the extra parameters passed to them after the help index parameter specify the valid range for the field and, in the case of AddRealField, the maximum number of decimal places to be displayed (unnecessary here since the picture masks have fixed decimal points). The "missing" parameter in the calls to AddByteField and AddRealField is the field width. This information isn't needed, because it is supplied by the (required) picture masks.

The last field in the entry screen is designated as a protected field by activating the efProtected option just before it is added. This is done because the field is a "calculated field," meaning that its value is always computed as a function of the Wage field, which represents the amount of money earned per hour. This computation is done in the post-edit routine, which is activated by calling SetPostEditProc. A post-edit routine is a user-written procedure that will be called immediately after one of the fields in the entry screen has been edited. Post-edit routines and calculated fields are discussed in detail in a later section.

The call to SetPreEditProc activates a pre-edit routine, which will be called just before a field is edited. The typical use for such a routine is to display a help message that provides the user with any special instructions relating to the field. Pre-edit routines are also discussed more fully in a later section.

Now we're ready to start editing. As with most objects derived from CommandWindow, the call to Process is part of a repeat..until loop that includes a case statement in which exit commands are handled. The only ones likely to occur in our example program are ccDone (<CtrlEnter>), ccQuit (<Esc>), and ccError (fatal error). But there are several other possibilities, especially if you have enabled any of the user exit commands.

The program ends by erasing the EntryScreen and destroying it. The call to Done is unnecessary in this case, since the memory allocated to the EntryScreen will be recovered by DOS when the program ends, but it would be needed if the program were going to continue.

Now that you've "written" your first program with OPENTRY, where do you go from here? The first thing you should do is read the remainder of this introduction, which covers the topics we've already mentioned in more detail, as well as a few that we haven't touched on at all. Don't worry if you can't digest all the information the first time through. You can always return to it later.

The last section of the introduction contains an overview of the entire data entry system implemented in OPENTRY. This is intended to help you see how all the pieces fit together. Although you might be tempted to skip ahead and read it right now, we recommend that you wait until you've read the sections that precede it, since parts of the overview assume familiarity with them.

Your next step should probably be to read the documentation for the MAKESCRN utility, which allows you to design entry screens interactively. Besides being an invaluable time-saver, MAKESCRN is also a good teaching tool, because it generates efficient, readable source code.

After you've experimented with MAKESCRN a bit, and you feel sufficiently comfortable with the basic concepts presented here, you should look at ENTRY.PAS. This demonstration program was designed to demonstrate virtually all of the major features of OPENTRY, as well as some of the minor ones.

Adding Fields to an EntryScreen

Because there are roughly 50 methods for adding fields to an EntryScreen, they are discussed here in groups according to the field editor that they use, rather than individually. For a discussion of field editors in general, and descriptions of the behavior of specific field editors, see the documentation for the OPFEDIT unit.

There is relatively little variation in the parameters passed to these methods, and once you've learned how to use one of them it is very easy to learn how to use the others, especially those in the same group.

The first three items in the parameter lists for all these methods are the same: Prompt, pRow, and pCol. The Prompt, if any, is a brief message to be displayed near the actual field, usual just in front of it. pRow and pCol give the coordinates for the Prompt. In the case of a regular EntryScreen, these coordinates are always construed to be relative to the top left corner of the EntryScreen's inner window. (Field coordinates in a ScrollingEntryScreen, discussed in the next section, are interpreted a bit differently.)

These three parameters are usually followed by three more that pertain to the field itself: Picture (or PicChar), fRow, and fCol. fRow and fCol, as you might expect, are the coordinates for the field itself. Picture is the picture mask for the field, which describes the format of the field and imposes limits on the characters that can be entered at a given position in the field. (For a discussion of picture masks, see the OPABSFLD chapter.) In a few cases, where there is a reasonable default, the Picture parameter is optional. These cases are addressed individually below. If the comments following the declaration for a given method do not indicate that there is a default picture mask, there isn't one.

If the field is a "simple" field, the Picture parameter's place is usually taken by a PicChar parameter. PicChar is a single picture mask character–such as 'X', '!', '9', '#'–that indicates what characters may legally be entered at any position in the field. In a couple of cases (a simple boolean field, for example), there is only one possible value for the PicChar parameter, so it is assumed.

The last parameter to an AddField method (EditXxx) is almost always the name of the variable associated with the field. The only exceptions are AddNestedField (not associated with any variable) and AddPickStringField (pick list is last parameter, string variable is next-to-last).

Beyond that, there is a fair amount of variation in the parameters passed to the various AddField methods. The order of parameters is consistent, however. It just looks like it isn't because not all parameters are required for all fields. Here is the basic order of parameters and the types of fields that require them:

Parameter	Meaning	Field Types
Prompt	Prompt message	All
pRow, pCol	Prompt coordinates	All
Picture/PicChar	Picture mask/char	Most
fRow, fCol	Field coordinates	All
fWidth	Field width	String, array, most simple, user-defined
fHeight	Field height	Multi-line only
MaxLen	Maximum length	Simple string only
HelpIndex	Help index	All
XxxLo, XxxHi	Lower, upper limits	Char, numbers, Date, Time
DataSize	Size of variable	Multiple choice, user-defined
Increment	IncChoice routine	Multiple choice only
Places	Decimal places	All reals except Comp
Validate	Validation routine	User-defined only
Convert	Conversion routine	User-defined only
FDraw	Drawing routine	User-defined only
Editor	Field editor	User-defined only
EditXxx	Variable to edit	Most
PL	Pick list	Pick field only

Let's talk briefly about the more common of the parameters not discussed already.

fWidth is the width of the field. In most cases, the length of the Picture mask conveys this information. But in the cases of most simple field types, where there is no true picture mask, the width of the field cannot be inferred. Also, in the cases of strings and arrays of char, the field width may be less than the maximum length of the string/array (indicated by the length of the picture mask or the MaxLen parameter), to allow the field to be scrolled horizontally. For example, a string field might have a maximum length of 254 (the absolute maximum), but a width of 40. If the user enters more than 40 characters in the string, the field will be scrolled horizontally to allow characters beyond the 40th position to be displayed and edited.

The HelpIndex parameter specifies the value to be passed to a user-defined help routine if the ccHelp command (<F1>) is issued while that field is being edited.

The XxxLo and XxxHi parameters, required for all fields of a numeric type (Integer, Real, Date, etc.), specify the lower and upper limits for values entered into the field. If the two parameters are equal, the user can enter any legal value in the field. For example, if the field is of type Byte, and the lower and upper limits are both 0, the valid range for the field will be 0-255.

The Places parameter is required for all real fields (Real, BCD, Single) except those associated with a variable of type Comp, which has no fractional part. If the picture mask for the field has a fixed decimal point, Places is ignored. If it doesn't, Places indicates the maximum number of decimal places to be displayed; the user can enter more, but they will be ignored when the cursor leaves the field. For example, let's say the user enters '1.2345' and Places is 2: the field will show only '1.23'. If the user instead enters '1.20', only '1.2' will be displayed--trailing 0's will be suppressed. If the user enters '1.201', the result will also be '1.2', because a maximum of two decimal places can be displayed and the second decimal place holds 0.

Note that the Places parameter is also currently ignored for numeric real fields because the NumberEditor cannot deal with a "floating" decimal point. If you want, say, two decimal places to be displayed, you must have a fixed decimal point in the picture mask. Future versions of OPENTRY may have use for the Places parameter, however.

The remaining parameters shown in the table are required for only one or two of the AddField routines, and so are discussed individually below.

StringEditor
The following methods all accept true picture masks as parameters.

```
procedure AddStringField(Prompt : string;  pRow, pCol : Word;
                         Picture : string; fRow, fCol : Word;
                         fWidth : Byte;    HelpIndex : Word;
                         var EditSt : string);
  {-Add a field of type string}
procedure AddArrayField(Prompt : string;  pRow, pCol : Word;
                        Picture : string; fRow, fCol : Word;
                        fWidth : Byte;    HelpIndex : Word;
                        var EditVar);
  {-Add a field of type array (of char)}
```

fWidth is the width of the field. If it is less than Length(Picture), the field will be scrolled horizontally as necessary during editing. If Picture is empty, a picture mask of fWidth 'X' characters will be assumed.

```
procedure AddLongField(Prompt : string;  pRow, pCol : Word;
                       Picture : string; fRow, fCol : Word;
                       HelpIndex : Word; LongLo, LongHi : LongInt;
                       var EditLong : LongInt);
  {-Add a field of type LongInt}
procedure AddWordField(Prompt : string;  pRow, pCol : Word;
                       Picture : string; fRow, fCol : Word;
                       HelpIndex : Word; WordLo, WordHi : Word;
                       var EditWord : Word);
  {-Add a field of type word}
procedure AddIntField(Prompt : string;  pRow, pCol : Word;
                      Picture : string; fRow, fCol : Word;
                      HelpIndex : Word; IntLo, IntHi : Integer;
                      var EditInt : Integer);
  {-Add a field of type integer}
procedure AddByteField(Prompt : string;  pRow, pCol : Word;
                       Picture : string; fRow, fCol : Word;
                       HelpIndex : Word; ByteLo, ByteHi : Byte;
                       var EditByte : Byte);
  {-Add a field of type byte}
procedure AddShortField(Prompt : string;   pRow, pCol : Word;
                        Picture : string;  fRow, fCol : Word;
                        HelpIndex : Word;  ShortLo, ShortHi : ShortInt;
                        var EditShort : ShortInt);
  {-Add a field of type shortint}
procedure AddRealField(Prompt : string;   pRow, pCol : Word;
                       Picture : string;  fRow, fCol : Word;
                       HelpIndex : Word;  RealLo, RealHi : Real;
                       Places : Byte;     var EditReal : Real);
  {-Add a field of type real}
```

```
{$IFDEF UseBcd}
procedure AddBcdField(Prompt : string;   pRow, pCol : Word;
                      Picture : string;  fRow, fCol : Word;
                      HelpIndex : Word;  BcdLo, BcdHi : BCD;
                      Places : Byte;     var EditBcd : BCD);
  {-Add a field of type BCD}
{$ENDIF}

{$IFOPT N+}
procedure AddExtField(Prompt : string;  pRow, pCol : Word;
                      Picture : string; fRow, fCol : Word;
                      HelpIndex : Word; ExtLo, ExtHi : Extended;
                      Places : Byte;    var EditExt : Extended);
  {-Add a field of type extended}

procedure AddDblField(Prompt : string;  pRow, pCol : Word;
                      Picture : string; fRow, fCol : Word;
                      HelpIndex : Word; DblLo, DblHi : Double;
                      Places : Byte;    var EditDbl : Double);
  {-Add a field of type double}

procedure AddSglField(Prompt : string;  pRow, pCol : Word;
                      Picture : string; fRow, fCol : Word;
                      HelpIndex : Word; SglLo, SglHi : Single;
                      Places : Byte;    var EditSgl : Single);
  {-Add a field of type single}

procedure AddCompField(Prompt : string;  pRow, pCol : Word;
                       Picture : string; fRow, fCol : Word;
                       HelpIndex : Word; CompLo, CompHi : Comp;
                       var EditComp : Comp);
  {-Add a field of type comp}
{$ENDIF}

{$IFDEF UseDates}
procedure AddDateField(Prompt : string;      pRow, pCol : Word;
                       Picture : DateString; fRow, fCol : Word;
                       HelpIndex : Word;     DateLo, DateHi : Date;
                       var EditDate : Date);
  {-Add a field of type date}

procedure AddDateStField(Prompt : string;      pRow, pCol : Word;
                         Picture : DateString; fRow, fCol : Word;
                         HelpIndex : Word;     var EditDate : DateString);
  {-Add a field of type date string}

procedure AddTimeField(Prompt : string;      pRow, pCol : Word;
                       Picture : DateString; fRow, fCol : Word;
                       HelpIndex : Word;     TimeLo, TimeHi : Time;
                       var EditTime : Time);
  {-Add a field of type time}
{$ENDIF}
```

The XxxLo and XxxHi parameters specify the lower and upper limits for the number being edited (last parameter). If XxxHi = XxxLo, any value that is valid for the given data type may be entered.

```
procedure AddNestedField(Prompt : string;  pRow, pCol : Word;
                         Picture : string; fRow, fCol : Word;
                         fWidth : Byte;    HelpIndex : Word);
  {-Add a dummy field that is associated with a nested entry screen}
```

```
procedure AddNestedStringField(Prompt : string;  pRow, pCol : Word;
                               Picture : string; fRow, fCol : Word;
                               fWidth : Byte;    HelpIndex : Word;
                               var EditSt : string);
  {-Add a string field that is associated with a nested entry screen}
```

These two methods are discussed a bit later in the section on "Nested Data Entry Screens."

SimpleStringEditor

The following methods use the simple string editor, which does not work with true picture masks. Note that all of them are affected by the leHouseCursorAtEnd option. If it is on, the width of the edit field is effectively increased by one to make a place to "house" the cursor when the field is full.

```
procedure AddSimpleStringField(Prompt : string;       pRow, pCol : Word;
                               PicChar : Char;        fRow, fCol : Word;
                               fWidth, MaxLen : Byte; HelpIndex : Word;
                               var EditSt : string);
  {-Add a field of type string (simple)}
```

MaxLen gives the maximum length of the string to be edited (EditSt), and fWidth specifies the width of the editing field. If MaxLen > fWidth, the field will be scrolled horizontally as necessary.

```
procedure AddSimpleLongField(Prompt : string;    pRow, pCol : Word;
                             PicChar : Char;     fRow, fCol : Word;
                             fWidth : Byte;      HelpIndex : Word;
                             LongLo, LongHi : LongInt;
                             var EditLong : LongInt);
  {-Add a field of type LongInt (simple)}
procedure AddSimpleWordField(Prompt : string;    pRow, pCol : Word;
                             PicChar : Char;     fRow, fCol : Word;
                             fWidth : Byte;      HelpIndex : Word;
                             WordLo, WordHi : Word;
                             var EditWord : Word);
  {-Add a field of type word (simple)}
procedure AddSimpleIntField(Prompt : string;    pRow, pCol : Word;
                            PicChar : Char;     fRow, fCol : Word;
                            fWidth : Byte;      HelpIndex : Word;
                            IntLo, IntHi : Integer;
                            var EditInt : Integer);
  {-Add a field of type integer (simple)}
procedure AddSimpleByteField(Prompt : string;    pRow, pCol : Word;
                             PicChar : Char;     fRow, fCol : Word;
                             fWidth : Byte;      HelpIndex : Word;
                             ByteLo, ByteHi : Byte;
                             var EditByte : Byte);
  {-Add a field of type byte (simple)}
procedure AddSimpleShortField(Prompt : string;    pRow, pCol : Word;
                              PicChar : Char;     fRow, fCol : Word;
                              fWidth : Byte;      HelpIndex : Word;
                              ShortLo, ShortHi : ShortInt;
                              var EditShort : ShortInt);
  {-Add a field of type shortint (simple)}
```

The XxxLo and XxxHi parameters specify the lower and upper limits for the number being edited.

```
procedure AddSimpleRealField(Prompt : string;       pRow, pCol : Word;
                             PicChar : Char;        fRow, fCol : Word;
                             fWidth : Byte;         HelpIndex : Word;
                             RealLo, RealHi : Real; Places : Byte;
                             var EditReal : Real);
  {-Add a field of type real (simple)}
{$IFDEF UseBcd}
procedure AddSimpleBcdField(Prompt : string;    pRow, pCol : Word;
                            PicChar : Char;     fRow, fCol : Word;
                            fWidth : Byte;      HelpIndex : Word;
                            BcdLo, BcdHi : BCD; Places : Byte;
                            var EditBcd : BCD);
  {-Add a field of type BCD (simple)}
{$ENDIF}
{$IFOPT N+}
procedure AddSimpleExtField(Prompt : string;         pRow, pCol : Word;
                            PicChar : Char;          fRow, fCol : Word;
                            fWidth : Byte;           HelpIndex : Word;
                            ExtLo, ExtHi : Extended; Places : Byte;
                            var EditExt : Extended);
  {-Add a field of type extended (simple)}
procedure AddSimpleDblField(Prompt : string;       pRow, pCol : Word;
                            PicChar : Char;        fRow, fCol : Word;
                            fWidth : Byte;         HelpIndex : Word;
                            DblLo, DblHi : Double; Places : Byte;
                            var EditDbl : Double);
  {-Add a field of type double (simple)}
procedure AddSimpleSglField(Prompt : string;       pRow, pCol : Word;
                            PicChar : Char;        fRow, fCol : Word;
                            fWidth : Byte;         HelpIndex : Word;
                            SglLo, SglHi : Single; Places : Byte;
                            var EditSgl : Single);
  {-Add a field of type single (simple)}
procedure AddSimpleCompField(Prompt : string;       pRow, pCol : Word;
                             PicChar : Char;        fRow, fCol : Word;
                             fWidth : Byte;         HelpIndex : Word;
                             CompLo, CompHi : Comp; var EditComp : Comp);
  {-Add a field of type comp (simple)}
{$ENDIF}
```

Places indicates the maximum number of decimal places to be displayed initially. Note that, if PicChar = 'E', values may be entered using scientific notation.

CharEditor

The methods shown below use CharEditor for editing. The picture mask passed to them should contain a single formatting character, one that is appropriate for the field type ('Y' for AddYesNoField, for example). It may also contain any number of literal characters (e.g., '>Y<'). In the case of AddCharField, the CharLo and CharHi parameters specify the lower and upper limits for the character entered. If CharHi = CharLo, the picture mask alone determines what characters may be entered. Otherwise, they impose further limits on the range of characters that may be entered.

```
procedure AddCharField(Prompt : string;  pRow, pCol : Word;
                       Picture : string; fRow, fCol : Word;
                       HelpIndex : Word; CharLo, CharHi : Char;
                       var EditChar : Char);
  {-Add a field of type char}
```

```
procedure AddBooleanField(Prompt : string;  pRow, pCol : Word;
                          Picture : string; fRow, fCol : Word;
                          HelpIndex : Word; var EditBool : Boolean);
  {-Add a field of type boolean}
procedure AddYesNoField(Prompt : string;  pRow, pCol : Word;
                        Picture : string; fRow, fCol : Word;
                        HelpIndex : Word; var EditYesNo : Boolean);
  {-Add a field of type yes/no}
```

Each of these methods has a default picture mask that is used if Picture = ''. For AddCharField, it is 'X' (AnyChar). For AddBooleanField, it is 'B' (BooleanOnly). And for AddYesNoField it is 'Y' (YesNoOnly).

SimpleCharEditor

The following methods use the simple character editor, which does not work with true picture masks.

```
procedure AddSimpleCharField(Prompt : string;    pRow, pCol : Word;
                             PicChar : Char;     fRow, fCol : Word;
                             HelpIndex : Word;   CharLo, CharHi : Char;
                             var EditChar : Char);
  {-Add a field of type char (simple)}
```

PicChar indicates how the case of the character entered should be changed, if at all, and sets limits on the range of characters that may be entered. If CharHi = CharLo, the picture char alone determines what characters may be entered. Otherwise, they impose further limits on the range of characters that may be entered.

```
procedure AddSimpleBooleanField(Prompt : string;   pRow, pCol : Word;
                                fRow, fCol : Word; HelpIndex : Word;
                                var EditBool : Boolean);
  {-Add a field of type boolean (simple)}
procedure AddSimpleYesNoField(Prompt : string;   pRow, pCol : Word;
                              fRow, fCol : Word; HelpIndex : Word;
                              var EditYesNo : Boolean);
  {-Add a field of type yes/no (simple)}
```

No PicChar parameter is required here because the only valid values would be 'B' and 'Y', respectively.

NumberEditor

The following methods allow you to edit numeric variables using the NumberEditor. The Places parameter for the methods involving reals is currently ignored. If fractional numbers are to be allowed, the picture mask must contain a fixed decimal point. Future versions of **Object Professional** may make use of the Places parameter.

```
procedure AddNumericLongField(Prompt : string;  pRow, pCol : Word;
                              Picture : string; fRow, fCol : Word;
                              HelpIndex : Word; LongLo, LongHi : LongInt;
                              var EditLong : LongInt);
  {-Add a field of type LongInt (numeric)}
procedure AddNumericWordField(Prompt : string;  pRow, pCol : Word;
                              Picture : string; fRow, fCol : Word;
                              HelpIndex : Word; WordLo, WordHi : Word;
                              var EditWord : Word);
  {-Add a field of type word (numeric)}
```

```
procedure AddNumericIntField(Prompt : string;  pRow, pCol : Word;
                             Picture : string; fRow, fCol : Word;
                             HelpIndex : Word; IntLo, IntHi : Integer;
                             var EditInt : Integer);
  {-Add a field of type integer (numeric)}
procedure AddNumericByteField(Prompt : string;  pRow, pCol : Word;
                              Picture : string; fRow, fCol : Word;
                              HelpIndex : Word; ByteLo, ByteHi : Byte;
                              var EditByte : Byte);
  {-Add a field of type byte (numeric)}
procedure AddNumericShortField(Prompt : string;   pRow, pCol : Word;
                               Picture : string;  fRow, fCol : Word;
                               HelpIndex : Word;  ShortLo, ShortHi : ShortInt;
                               var EditShort : ShortInt);
  {-Add a field of type shortint (numeric)}
procedure AddNumericRealField(Prompt : string;  pRow, pCol : Word;
                              Picture : string; fRow, fCol : Word;
                              HelpIndex : Word; RealLo, RealHi : Real;
                              Places : Byte;    var EditReal : Real);
  {-Add a field of type real (numeric)}
{$IFDEF UseBcd}
procedure AddNumericBcdField(Prompt : string;  pRow, pCol : Word;
                             Picture : string; fRow, fCol : Word;
                             HelpIndex : Word; BcdLo, BcdHi : BCD;
                             Places : Byte;    var EditBcd : BCD);
  {-Add a field of type BCD (numeric)}
{$ENDIF}
{$IFOPT N+}
procedure AddNumericExtField(Prompt : string;  pRow, pCol : Word;
                             Picture : string; fRow, fCol : Word;
                             HelpIndex : Word; ExtLo, ExtHi : Extended;
                             Places : Byte;    var EditExt : Extended);
  {-Add a field of type extended (numeric)}
procedure AddNumericDblField(Prompt : string;  pRow, pCol : Word;
                             Picture : string; fRow, fCol : Word;
                             HelpIndex : Word; DblLo, DblHi : Double;
                             Places : Byte;    var EditDbl : Double);
  {-Add a field of type double (numeric)}
procedure AddNumericSglField(Prompt : string;  pRow, pCol : Word;
                             Picture : string; fRow, fCol : Word;
                             HelpIndex : Word; SglLo, SglHi : Single;
                             Places : Byte;    var EditSgl : Single);
  {-Add a field of type single (numeric)}
procedure AddNumericCompField(Prompt : string;  pRow, pCol : Word;
                              Picture : string; fRow, fCol : Word;
                              HelpIndex : Word; CompLo, CompHi : Comp;
                              var EditComp : Comp);
  {-Add a field of type comp (numeric)}
{$ENDIF}
```

Note that none of these fields has a default picture mask.

Others

The following methods do not fit into any of the general categories above, usually because they use a special-purpose field editor.

```
procedure AddChoiceField(Prompt : string;  pRow, pCol : Word;
                         Picture : string; fRow, fCol : Word;
                         HelpIndex : Word; DataSize : Byte;
                         Increment : IncChoiceProc;
                         var EditVar);
  {-Add a multiple choice field}
```

This method is used for fields of enumerated types and more generally for any data type of up to 10 bytes that can be incremented and decremented. DataSize is the size of the variable, and Increment is a user-written IncChoice routine. See "Multiple Choice Fields," below.

```
procedure AddMultiLineField(Prompt : string;        pRow, pCol : Word;
                            PicChar : Char;         fRow, fCol : Word;
                            fWidth, fHeight : Byte; HelpIndex : Word;
                            var EditVar);
  {-Add a multi-line field}
```

This method allows you to edit an array of strings. EditVar is assumed to be an 'array[1..fHeight] of string[fWidth]'. See the discussion of "Multi-line Fields," below, for details.

```
{$IFDEF PickListFields}
procedure AddPickStringField(Prompt : string;     pRow, pCol : Word;
                             fRow, fCol : Word;   fWidth : Byte;
                             HelpIndex : Word;    var EditSt : string;
                             var PL : PickList);
  {-Add a field of type string to be edited with a pick list}
{$ENDIF}
```

This method allows you to "edit" a string variable (EditSt) using a PickList (PL), which is invoked automatically each time the cursor is moved to the field. See the discussion of "Pick Fields," below, for details.

```
procedure AddUserField(Prompt : string;     pRow, pCol : Word;
                       Picture : string;    fRow, fCol : Word;
                       fWidth : Byte;       HelpIndex : Word;
                       RangeLo : RangeType; RangeHi : RangeType;
                       DataSize : Word;     DecimalPlaces : Byte;
                       Validate: ValidationFunc;
                       Convert : ConversionProc;
                       FDraw   : DrawProc;
                       Editor  : EditProc;
                       var EditVar);
  {-Add a field of a user-defined type}
```

This method allows you to edit a variable of a "user-defined" type (one not supported directly by OPENTRY). See the discussion of "User-Defined Field Types," below, for details.

Field Options

The various options that apply to the fields in a data entry screen are discussed in the documentation for the EntryField type in the OPFIELD section. If you haven't done so already, you should be sure to read that discussion. If you have, you know that the options are divided somewhat arbitrarily into two categories: primary EntryField options ('ef' prefix) and Secondary EntryField options ('sef' prefix). We won't repeat the descriptions given there, but we do want to call your attention to some options that either have special meaning for fields in an entry screen or require special comment in this context.

The efClickExit option allows you to indicate that, if a user issues the ccMouseSel command on a given field, the EntryScreen should exit immediately, and the GetLastCommand function should return ccClickExit. This option is useful in cases where the user can either enter a

response manually or, if he issues a user exit command, choose a response from a pick list. In such a case, ccClickExit could be treated as equivalent to the user exit command.

The efExitAtEdges option pertains only to pick fields, which are discussed more fully later. It indicates that the EntryScreen should automatically activate the pkExitAtEdges option for the PickList before calling it.

The efAllowEscape option pertains only to pick fields, discussed later, and simple fields that use the SimpleCharEditor (simple Char, Boolean, and yes-no). This option should generally be turned off for pick fields, and it should definitely be on (as it is by default) in all other cases.

The efRequired option allows you to designate one or more fields as being required–meaning that they cannot be left blank. Required fields are also discussed more fully below.

The secondary field option sefPadCurrentOnly indicates that the special pad character for a field should be used only when that field is being edited. You might, for example, want to set the pad character to '_' for all of the fields, but have the underscores displayed only during editing. The sefPadCurrentOnly option lets you do this.

Field Delimiters

Many programmers (and users) prefer to have each field in a data entry screen surrounded–or "delimited"–by a pair of characters that mark the boundaries of the field. For example:

```
Name     [John Doe          ]
Address  [123 Main St.      ]
City     [Anytown           ]
State    [US]
Zip      [12345]
```

This is accomplished in OPENTRY by calling, SetDelimiters, which affects all fields in the entry screen. For example, to surround the fields with brackets, you would execute

```
ES.SetDelimiters('[', ']');
```

which sets the left delimiter to '[' and the right delimiter to ']'. The default left and right delimiters are both null characters (#0). *If either of the two delimiters is a null, neither delimiter will be displayed.*

Note that, if you want your EntryScreen to use delimiters, you should call SetDelimiters *before* adding any fields to the EntryScreen. If you wait until later to call it, the routines that add the fields will not be able to validate their coordinates and dimensions properly.

Pre-Edit Routines

A pre-edit routine is a user-written procedure called by Process just before the field editor is called to edit the current field. The most common use for this hook is to display a help message concerning the field to be edited, but it can also be used to customize the behavior of the EntryScreen. For example, it can force the EntryScreen to execute a particular command by calling SetLastCommand, or it can move the cursor to a different field by calling SetNextField.

A pre-edit routine has the following form:

```
{$F+}
procedure PreEdit(ESP : EntryScreenPtr);
begin
  case ESP^.GetCurrentID of
    {display help message for this field}
  end;
end;
{$F-}
...
ES.SetPreEditProc(PreEdit);
```

The ESP parameter is a pointer to the EntryScreen making the call.

For an example of a routine that displays help messages using this hook, see ENTRY.PAS.

Post-Edit Routines and Calculated Fields

The counterpart of the pre-edit routine is the post-edit routine, a user-written procedure that is called by Process immediately after the user has edited the current field. A post-edit routine has the following form:

```
{$F+}
procedure PostEdit(ESP : EntryScreenPtr);
begin
  case ESP^.GetCurrentID of
    {update any fields affected by changes to the current field}
    {update the screen using DrawField}
  end;
end;
{$F-}
...
ES.SetPostEditProc(PostEdit);
```

The ESP parameter is a pointer to the EntryScreen making the call.

Like the pre-edit routine, the post-edit routine can be used to customize the behavior of the EntryScreen. For example, it can force the EntryScreen to execute a command other than the one issued by the user by calling SetLastCommand, or it can force the cursor to move to a different field by calling SetNextField (after using 'SetLastCommand(ccNone)' to cancel the user's command).

But the more common use for a post-edit routine is to implement "calculated fields." A calculated field is a field whose value is a function of the value of one or more of the other fields in the entry screen. For example, if one of the fields contains an employee's date of birth (a value that must be entered by the user), a second field might contain the employee's age (a value that can be calculated based on the difference between the date of birth and the current date).

A calculated field is typically, but not necessarily, a protected field whose value is calculated once before the entry screen is first drawn, and then recalculated each time that the user edits one of the fields that its value is based on. For example, let's say that the value for field 7 is a function of the value for field 6. Here's a rough outline of a post-edit routine that would keep field 7 updated:

```
procedure PostEdit(ESP : EntryScreenPtr);
begin
  case ESP^.GetCurrentID of
    6 : begin
          {recalculate value for field 7}
          ESP^.DrawField(7);
        end;
  end;
end;
```

The post-edit always checks the ID number for the current field. If field 6 is the current field (meaning that it was just edited), the value for field 7 is recalculated. DrawField, a method inherited from AbstractSelector, is then called to redraw field 7.

For a simple example of a post-edit routine, see the short program at the beginning of this section. For a more elaborate example, see ENTRY.PAS.

Multiple Choice Fields

As with calculated fields, OPENTRY requires some assistance in handling multiple-choice fields. When you call AddChoiceField to add such a field, it asks for the name of a routine (called an IncChoice routine) that can convert values to strings and increment or decrement a value after the user asks for another choice. In outline, such a routine would look something like this:

```
{$F+}
procedure IncChoice(var Value; ID : Word; Factor : Integer; var S : string);
begin
  case Factor of
    +1 : {increment Value} ;
    -1 : {decrement Value} ;
  end;
  S := {string representation of Value} ;
  {update calculated fields, if any}
end;
{$F-}
```

It is up to IncChoice to do whatever typecasting is necessary to work with Value. It can either use an explicit typecast, such as 'Byte(Value)', or it can declare an absolute variable of the appropriate type--'ByteValue : Byte absolute Value'--and work with it. Factor indicates whether Value needs to be incremented or decremented before converting it to a string; if the Factor is 0, only conversion is necessary. (Factor will always be in the range -1..1.) ID indicates which field in an entry screen Value is associated with, making it possible to write a single routine that handles all multiple-choice fields in a given entry screen.

Although this scheme was designed to handle enumerated types such as 'Gender = (Male, Female, Unknown)', it can also be used for editing variables of numeric types (integer, real, double) that you want the user to change only in fixed increments. Neither AddChoiceField nor the multiple choice field editor knows or cares about the actual type of the variable to be edited. For example, you might have a field of type Integer that was incremented or decremented by multiples of 100 rather than 1 (100*Factor rather than 1*Factor). The only restriction they impose is that the variable can be no larger than 10 bytes.

One benefit of making a numeric variable a multiple choice field is that any calculated fields associated with it can be updated continuously. For example, an entry screen that contains data concerning a loan or mortgage might have two numeric multiple choice fields--one for the interest rate, and one for the term of the loan (in months or years)--and one or more calculated

fields based on the two: monthly payment, total interest paid over the life of the loan, etc. As the user changed the interest rate or the term of the loan, the calculated fields could be updated instantly to show the effects of the change. The way to accomplish this effect is to make the IncChoice routine, rather than the post-edit routine, update the calculated field(s).

The sample program ENTRY.PAS demonstrates both uses of multiple-choice fields (enumerated types and numeric types), and has an example of a continuously updated calculated field.

Special note to users of Turbo Professional's TPENTRY unit: In TPENTRY, the multiple choice editor called the entry screen's post-edit routine immediately after calling the IncChoice routine to allow for continuously updated calculated fields. This behavior simplified matters in some cases, and complicated them in others. In **Object Professional**, a post-edit routine is called only by the Process method.

Required Fields

Normally, any field in a data entry screen may be left empty. In some cases, however, empty fields are simply unacceptable, and in these situations you should designate the field as required. To do so, you just activate the efRequired option before adding such a field, then deactivate it afterward:

```
esFieldOptionsOn(efRequired);
{add one or more required fields}
esFieldOptionsOff(efRequired);
```

When a field is marked as required, its contents will be checked each time that it is edited, as well as when the user exits from the entry screen. If all of the non-literal characters in the field are blanks, the EntryScreen will generate an error. Note that no error will be reported if the field is partially full. If you need to be certain that a string field contains no blank spaces, you will need to specify a special validation routine, such as ValidateNoBlanks (in OPFIELD), by calling ChangeValidation after you add the field.

Required fields, we should point out, are checked on exit from Process only if a "hard" exit command was issued. The distinction between "hard" and "soft" exit commands is explained in the entry for the Process method.

Global Validation Routines

OPFIELD and OPENTRY provide several means for insuring that the data entered into a given field is valid. Picture masks ensure that only valid characters are entered into the field. Individual field validation routines can verify that the value entered is meaningful and that it is within an acceptable range of values. And the required field option can ensure that a given field is not left completely blank.

But in some cases these forms of validation are inadequate. For example, let's suppose that an entry screen has five byte fields that allow the user indicate an order of preference. The valid range for each field would be 1-5, but no two fields should have the same value. Until the user has edited all five fields, it would not be possible to validate any one of them. A post-edit routine could conceivably perform a consistency check after the fifth field was edited, but that would force the user to enter his preferences in a specific order, and it wouldn't account for the possibility that he might go back and change one of his earlier responses. If you use a global validation routine to perform the consistency check, however, the work doesn't have to be done until the user gives a "hard" exit command (see the entry for Process).

A global validation routine must be of the following form:

```
{$F+}
function ValidateFunc(ESP : EntryScreenPtr; var EFP : EntryFieldPtr;
                      var ErrCode : Word; var ErrorSt : StringPtr) : Boolean;
begin
end;
```

ESP is a pointer to the entry screen whose contents are to be validated. If a problem is found, the function must return False and set the three var parameters as follows: EFP must point to the field that caused the error, ErrCode should contain the code for the error, and ErrorSt must point to a message that describes the error. The proper value for EFP may be obtained by using the expression

```
EFP := EntryFieldPtr( ESP^.FindField(ID) );
```

where ID is the field containing the error. If ValidateFunc returns False, the caller (the Process method) will position the cursor on the designated field and then pass the ErrCode and ErrorSt parameters to the EntryScreen's error handler.

If you do not supply a global validation routine of your own, OPENTRY will default to using one that consists simply of a call to the RevalidateAll method. RevalidateAll simply re-executes all the validation routines for the individual fields, in case the default value for a field not edited by the user is an invalid one. RevalidateAll should generally be called from any global validation routine after it performs all its application-specific consistency checks.

For a simple example of a global validation routine, see the entry for the SetValidateFunc method. A more complex example may also be found in the source code for MAKESCRN (GlobalValidate, in MSMAIN.IN2).

Nested Data Entry Screens

OPENTRY provides two means of working with data entry screens that are too big to fit on the physical screen. One is the ScrollingEntryScreen, described in the next section. The other is a hook that allows you to create nested entry screens. Here's a quick example of what we mean: Say you have a large database record that contains several fields that store information about a person's employment history. Rather than crowding all these fields into the main entry screen, you might want to display a dummy field called "Employment history". When the user moves the cursor to that field and presses <Enter>, a second data entry screen, one containing all the fields relating to the subject, could be displayed in a second, smaller entry screen overlapping the first. After the user has finished entering data into the second entry screen, you could then resume editing in the main entry screen.

Creating nested data entry screens requires three steps. The first is to add the dummy field to the main entry screen using AddNestedField (or AddNestedStringField). The second is to create the secondary entry screen and to write a procedure that will handle the calls to Process for it. The third is to modify the code that processes the exit commands for the main entry screen. If the exit command is ccNested, you would call the procedure that you wrote in Step 2. For an example of how to do this, see the Phones field in ENTRY.PAS.

On screen, a "nested field" (a field linked to a nested entry screen) looks much like an empty string field, and to some extent it acts like one as well. If the field is wider than 1 character, the cursor can be moved around within the field, for example. But data cannot be entered into it: the "field" simply marks a point at which the user can jump to a second entry screen. A variation on the nested field is the "nested string field" (AddNestedStringField), which allows you to specify a string variable whose contents will be displayed within the field. This string

might contain some of the information found in the fields in the nested entry screen, for example.

We should also point out that the hook provided for nested entry screens could be used in other situations. For example, let's say that one field in a data record is a large buffer set aside for "Notes," and that you want to use OPMEMO for editing the notes. The best way to do this is to create a dummy Notes field in the entry screen using AddNestedField or AddNestedStringField, and to invoke the memo editor when the ccNested command is returned. The Notes field in ENTRY.PAS demonstrates this technique, as well as the use of AddNestedStringField.

Multi-line Fields

Memo fields created using OPMEMO and AddNestedField are ideal for some applications, but they are overkill in others. Suppose, for example, that you just want to allow the user to enter at most 5 lines of say 60 characters each. This is just the type of situation that the multi-line field was designed for.

Here's an abbreviated example of how you would use AddMultiLineField to handle it:

```
var
  Scrap : record
            ...
            Notes : array[1..5] of string[60];
          end;
begin
  ...
  ES.AddMultiLineField('Notes',          {prompt}
                       5, 1,             {prompt coordinates}
                       'X',              {picture mask character}
                       6, 1,             {field coordinates}
                       60,               {field width}
                       5,                {field height}
                       0,                {help index}
                       Scrap.Notes);     {variable to edit}
  ...
end;
```

Here the prompt for the field is displayed at the start of row 5, and the multi-line field is displayed in the first 60 columns of rows 6-10. Notice the correspondence between the field width and height and the declaration for Scrap.Notes. A multi-line field must be associated with an array of strings–more specifically, an 'array[1..FieldHeight] of string[FieldWidth]'.

For another example of the use of AddMultiLineField, consult the source code for the phone book window in DESKPOP (DESKMAIN.IN2).

Pick Fields

Suppose that one of the fields in an entry screen required the user to enter the name of a pizza topping, and that you wanted to allow the user to select the choice of topping from the pick list shown in the example program at the start of the OPPICK chapter. There are three ways that you could do this:

1. You could make the field a regular string field (using AddStringField) and assign a certain key, say <F2>, to a user exit command such as ccUser1. If the user pressed <F2>, and the pizza topping field was the current field, you would allow the user to make his choice from the pick list. He would also have the option of entering the name of a topping manually.

2. Again, you would use AddStringField to create the field, but you would also write a pre-edit routine that would check the current field ID each time it was called. If the pizza topping field was current, the pre-edit routine would invoke the PickList, update the variable that stored the choice made by the user, redraw the field, then use SetNextField to move the cursor to another field. This approach would cause the PickList to be invoked automatically, eliminating the need for the user exit command.
3. You could use AddPickStringField and pass it the name of your PickList. The pick list would be displayed automatically when the cursor was moved to the pizza topping field, with the highlight placed on the item corresponding to the default choice.

Obviously, unless you need to allow the user to enter a string not in the pick list, option 3) is the clear winner. Here's an example of how you would do it:

```
{$V-}
program Pizza; {EXENTRY2.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpPick, OpField, OpSelect, OpEntry;
const NumPizzaToppings = 5;
type  InfoRec = record
                  Name : String[30];
                  PTop : String[10];
                end;
var
  Info : InfoRec;
  ES : EntryScreen;
  PL : PickList;

  {$F+}
  procedure PizzaTopping(Item : Word; Mode : pkMode; var IType : pkItemType;
                         var IString : String; PickPtr : PickListPtr);
  const
    ToppingStrings : array[1..5] of string[9] = (
      'Pepperoni', 'Sausage', 'Mushrooms', 'Anchovies', 'Onions');
  begin
    IString := ToppingStrings[Item];
    if Mode <> pkSearch then IString := ' '+IString+' ';
  end;
  {$F-}

begin
  if not PL.InitCustom(35, 5, 45, 10, DefaultColorSet,
                   DefWindowOptions OR wBordered, 20, NumPizzaToppings,
                   PizzaTopping, PickVertical, SingleChoice) then
    Halt;

  if not ES.InitCustom(25, 5, 55, 11, DefaultColorSet,
                     DefWindowOptions+wBordered) then
    Halt;

  ClrScr;
  FillChar(Info, SizeOf(Info), 0);
  { ES.esFieldOptionsOn(efExitAtEdges); }
  ES.AddSimpleStringField('Name:', 2, 1, 'A', 2, 7, 10, 20, 0, Info.Name);
  ES.AddPickStringField('Pizza Topping', 3, 1, 3, 7, 10, 0, Info.PTop, PL);

  ES.Process;

  ES.Done;
  PL.Done;
end.
```

As you can see, a call to AddPickStringField is very much like any other call to an AddField method, except that the last parameter is the name of your pick list. If you run this program, you'll see that the PTop field looks like any other simple string field--until the cursor is moved to it. Then the PickList will pop up.

Be aware that a PickList invoked by an EntryScreen recognizes several commands that it normally wouldn't respond to: ccTab, ccBackTab, ccNextField, ccWordLeft, ccWordRight, ccPrevField, ccNextRec, ccPrevRec, ccFirstFld, ccLastFld, ccNested, and ccDone. So, for example, if you press <Tab> or <CtrlRight>, the cursor will move out of the pick list to the next field in the entry screen. And keep in mind especially that pressing <Esc> (ccQuit) or <CtrlEnter> (ccDone) will cause not only the PickList to exit but also the EntryScreen: *any exit command issued within the PickList will be passed on to the EntryScreen for processing.* If you would prefer that ccQuit restore the default choice and move the cursor to next field, you can deactivate the efAllowEscape option before adding the pick field. This behavior is generally preferable to the default behavior.

There is one special field option that pertains only to pick fields, efExitAtEdges. If this option is set, the field editor for the pick field will temporarily activate the pkExitAtEdges option for the PickList before calling its Process method. If this option is activated, pressing <Up> when the cursor is on the first item in the pick list will cause the cursor to jump out of the pick list to the field above the current one. Pressing <Down> while on the last item would move the cursor to the field below the current one. To get a better idea of what the resulting behavior is like, try the example program shown above and remove the braces around the call to esFieldOptionsOn.

efExitAtEdges should generally be used only for small pick lists that don't scroll. If the pick list scrolls, the user will have difficulty determining when a cursor movement command will move the cursor to another item in the pick list, and when it will move the cursor to another field in the entry screen.

Pick fields generally work best in a non-scrollable EntryScreen that cannot be moved by the user. If your entry screen can be moved, or if it is a ScrollingEntryScreen, then the chances are good that when your PickList is invoked it may appear at an inappropriate location on the screen. If you want the coordinates of the PickList to be adjusted based on the location of the entry screen and/or the corresponding field, then you will need to write a pre-edit routine that will call either AdjustWindow or MoveWindow to reposition the PickList. If the new coordinates need to be based on the location of the field rather than the entry screen, then you can call the GetFieldCoordinates method to obtain the current field's absolute coordinates.

For another example of a pick field, see ENTRY.PAS, which uses a pick list to "edit" the State field of the main entry screen.

User-Defined Field Types

As we've seen, OPENTRY supports an almost bewildering array of field types. Nevertheless, there are bound to be cases where none of the standard field types fits the bill. What can you do when you run into one of these situations? You have two choices.

For the OOP purist, the obvious solution is to implement a new field object derived from EntryField, and an object derived from EntryScreen that provides the necessary AddXxxField method.

But if you're not entirely comfortable with objects, or you'd like to save yourself some effort, there is a simpler solution: AddUserField. In most cases, this approach requires you to write

only two routines, a conversion routine and a validation routine. Chances are good that one of the existing draw routines and field editors can handle the tasks of drawing and editing your new field type.

Because the possibilities for user-defined types are almost endless, we cannot begin to explain how to handle every conceivable case. We can, however, give you a realistic example of a particular case that you can use as a model.

Suppose that you needed to allow the user to edit a byte variable using binary, rather than decimal or hexadecimal, notation. Here is how you would do it:

```
{$V-,F+}
program EntryExample; {EXENTRY3.PAS}
uses
  OpCrt, OpString, OpInline, OpRoot, OpCmd, OpFrame, OpWindow, OpAbsFld,
  OpField, OpFEdit, OpSelect, OpEntry;
type
  InfoRec = record
              Name : String[30];
              Bin : Byte;
            end;
const
  BinLo : RangeType = (rtByte : 0);
  BinHi : RangeType = (rtByte : 255);
var
  Info : InfoRec;
  ExitCommand : Word;
  ES : EntryScreen;
  Finished : Boolean;

  function Str2Bin(S : String; var B : Byte) : Boolean;
  const
    BinMag : array[1..8] of Byte = ($80,$40,$20,$10,$08,$04,$02,$01);
  var
    I, Bin : Byte;
  begin
    Str2Bin := False;
    Bin := 0;
    S := TrimSpaces(S);
    if S = '' then
      Exit;
    S := LeftPadCh(S, '0', 8);
    for I := 1 to 8 do
      if S[I] = '1' then
        Inc(Bin, BinMag[I])
      else if S[I] <> '0' then
        Exit;
    Str2Bin := True;
    B := Bin;
  end;

  function ValidateBin(EFP : EntryFieldPtr; var ErrCode : Word;
                       var ErrorSt : StringPtr) : Boolean;
  var
    B : Byte;
    S : string[80];
  begin
    ValidateBin := False;
    with EFP^ do begin
```

```
      StripPicture(efEditSt^, S);
      if not Str2Bin(S, B) then begin
        ErrCode := ecBadFormat;
        ErrorSt := @emInvalidNumber;
      end
      else if (B < efRangeLo.rtByte) or (B > efRangeHi.rtByte) then begin
        ErrCode := ecOutOfRange;
        ErrorSt := @emOutOfRange;
      end
      else
        ValidateBin := True
    end;
  end;

  procedure BinConversion(EFP : EntryFieldPtr; PostEdit : Boolean);
  var
    S : string[80];
  begin
    with EFP^ do
      if PostEdit then begin
        StripPicture(efEditSt^, S);
        if not Str2Bin(S, Byte(efVarPtr^)) then
          Byte(efVarPtr^) := 0;
      end
      else begin
        S := BinaryB(Byte(efVarPtr^));
        MergePicture(S, efEditSt^);
      end;
  end;

begin
  ClrScr;
  FillChar(Info, SizeOf(Info), 0);
  if not ES.InitCustom(25, 5, 55, 9, DefaultColorSet,
                       DefWindowOptions OR wBordered) then
    Halt;

  UserSet1 := ['0', '1', ' '];
  ForceCaseUser['1'] := NoChange;

  ES.esFieldOptionsOn(efClearFirstChar);
  ES.AddStringField('Name:', 2, 1, '', 2, 9, 20, 0, Info.Name);
  ES.AddUserField('Bin:', 3, 1,          {prompt, coordinates}
                  '11111111', 3, 9,      {picture mask, field coordinates}
                  8,                     {field width}
                  0,                     {help index}
                  BinLo, BinHi,          {range}
                  1,                     {data size}
                  0,                     {decimal places}
                  ValidateBin,           {validation routine}
                  BinConversion,         {conversion routine}
                  DrawString,            {standard drawing routine}
                  StringEditor,          {standard field editor}
                  Info.Bin);             {data being edited}

  Finished := False;
  repeat
    ES.Process;
    case ES.GetLastCommand of
      ccQuit, ccDone, ccError : Finished := True;
```

```
      {Process user exit commands}
    end;
  until Finished;
  ES.Done;
end. .
```

As you can see, the parameters passed to AddUserField are quite similar to those passed to AddByteField or AddRealField. The only "unusual" parameters are the ones specifying the size of the data type (1 in our case) and the names of the four driver routines to be called to do all the work (ValidateBin, BinConversion, DrawString, and StringEditor). The last two, DrawString and StringEditor, are standard driver routines found in OPFIELD and OPFEDIT, respectively. These are appropriate choices for almost any field that has a true picture mask. The other two driver routines are ones we had to write especially for this application.

BinConversion does one of two things, depending on whether or not the field has just been edited. If PostEdit is True, it strips the picture mask from the string displayed to the user and then tries to convert the resulting string (S) to a byte value. If PostEdit is False, it takes the current byte value (pointed to by EFP^.efVarPtr) and converts it to a string.

The validation routine, ValidateBin, checks two things. It first checks to see if the string entered by the user is of a valid form. It will report an error, for example, if the user enters '10 01' rather than '1001'. It then checks to see if the value is in the proper range (0-255 in our case). If both these checks succeed, it returns True.

For further details on conversion and validation routines, and how to write them, see the documentation for OPFIELD.

The one other thing worth noting here is that we've used a user-defined mask character ('1') to limit the range of characters that the user can enter. Doing so doesn't simplify the task of the validation routine in this case, although it might in others. It simply insures that the user is not allowed to enter anything completely nonsensical in the field. The worst he can do is insert spaces between two of the digits.

EntryScreens and Streams

For some applications, such as MAKESCRN, the ability to store an EntryScreen in a stream is manna from heaven. It is possible, for example, to create a complex EntryScreen once, store it in a stream, and reuse it later in multiple programs. Large applications that have many entry screens can save code space if they are stored in a Library and loaded from disk when needed.

The only bad news is that it takes a bit more effort to store an EntryScreen in a stream and reload it than it does for most other objects. There are two reasons for the added complexity: the registration of all the EntryFields in the field list, and the registration of the variables associated with the fields. We'll tackle the field list problem first.

Why does the field list pose a problem? Couldn't the registration routine for the EntryScreen simply register all the field types for you automatically? Yes, it could, but if it did so the compiler would think that every single field type was used in your program—even if only a handful were actually used—and consequently a tremendous amount of dead code would be linked into the program unnecessarily. The convenience of having the EntryScreen register all the field types for you, rather than manually registering only the types you actually use, would come at too high a price for most people. So you have to do it manually.

Each field type that you actually use must be registered before you can store your EntryScreen in a stream or reload it from the stream. We've done two things to simplify this task. First, we've put together a table (below) that shows the correspondences between the various

AddField methods and the stream registration routines that go with them. This table makes it fairly easy to come up with a list of registration routines to call. The second thing we've done is to provide a few routines that register multiple field types at once:

```
AllEntryFieldsStream
AllPictureFieldsStream
AllSimpleFieldsStream
AllNumericFieldsStream
```

AllEntryFieldsStream registers every field type there is. If you wish, you can just call this one routine and be done with it. But unless you really need all of the different field types–highly unlikely, since even MAKESCRN doesn't use all of them–doing so is unwise. The other three register only a subset of the complete group. The field types registered by each routine are shown in the table below. An 'x' under Picture means that a field type is registered by AllPictureFieldsStream. An 'x' under Simple means it is registered by AllSimpleFieldsStream. And an 'x' under Numeric means it is registered by AllNumericFieldsStream. Field types that do not have in an 'x' in one of these columns can be registered only by calling the routine in the Stream registration column or by calling AllEntryFieldsStream.

AddField routine	Stream registration	Picture	Simple	Numeric
AddStringField	StringFieldStream	x		
AddArrayField	ArrayFieldStream	x		
AddCharField	CharFieldStream	x		
AddBooleanField	BooleanFieldStream	x		
AddYesNoField	YesNoFieldStream	x		
AddLongField	LongIntFieldStream	x		
AddWordField	WordFieldStream	x		
AddIntField	IntegerFieldStream	x		
AddByteField	ByteFieldStream	x		
AddShortField	ShortIntFieldStream	x		
AddRealField	RealFieldStream	x		
AddBcdField	BcdFieldStream	x		
AddExtField	ExtendedFieldStream	x		
AddDblField	DoubleFieldStream	x		
AddSglField	SingleFieldStream	x		
AddCompField	CompFieldStream	x		
AddDateField	DateFieldStream	x		
AddDateStField	DateStFieldStream	x		
AddTimeField	TimeFieldStream	x		
AddSimpleStringField	StringFieldSimStream		x	
AddSimpleCharField	CharFieldSimStream		x	
AddSimpleBooleanField	BooleanFieldSimStream		x	
AddSimpleYesNoField	YesNoFieldSimStream		x	
AddSimpleLongField	LongIntFieldSimStream		x	
AddSimpleWordField	WordFieldSimStream		x	
AddSimpleIntField	IntegerFieldSimStream		x	
AddSimpleByteField	ByteFieldSimStream		x	
AddSimpleShortField	ShortIntFieldSimStream		x	
AddSimpleRealField	RealFieldSimStream		x	
AddSimpleBcdField	BcdFieldSimStream		x	
AddSimpleExtField	ExtendedFieldSimStream		x	
AddSimpleDblField	DoubleFieldSimStream		x	
AddSimpleSglField	SingleFieldSimStream		x	
AddSimpleCompField	CompFieldSimStream		x	
AddNumericLongField	LongIntFieldNumStream			x
AddNumericWordField	WordFieldNumStream			x
AddNumericIntField	IntegerFieldNumStream			x
AddNumericByteField	ByteFieldNumStream			x
AddNumericShortField	ShortIntFieldNumStream			x
AddNumericRealField	RealFieldNumStream			x
AddNumericBcdField	BcdFieldNumStream			x
AddNumericExtField	ExtendedFieldNumStream			x
AddNumericDblField	DoubleFieldNumStream			x
AddNumericSglField	SingleFieldNumStream			x
AddNumericCompField	CompFieldNumStream			x
AddNestedField	NestedFieldStream			
AddNestedStringField	StringFieldStream	x		
AddChoiceField	ChoiceFieldStream			
AddMultiLineField	MultiLineFieldStream			
AddPickStringField	PickFieldStream			

The second problem involves the registration of all the variables that correspond to one of the fields in the EntryScreen. In the worst case, if no other solution were provided, you would have to register the address of each and every one of them using RegisterPointer. And you'd have to do it twice: once when the EntryScreen was stored, and once when it was loaded. Although it is

not possible to automate this task completely, the EntryScreen is able to simplify it for you using the concept of the "user record." The assumption behind the user record concept is that, in the typical data entry screen, each EntryField corresponds to a field in a single record variable. If this is so, then the matter of registering the variables can be made much simpler.

Before trying to store your EntryScreen, you can just call the SetUserRecord method with two parameters: the name and size of your record variable. When a given field is written to the stream, the Store method will see that the variable is part of the user record and simply save its offset relative to the start of the record. When the EntryScreen is reloaded from the stream, all that the Load method needs to know in order to fix up the addresses for all the variables is the address for the new user record. Here you have two choices:

1) If you do not register the address of the user record when you Store the EntryScreen, the Load constructor will try to allocate one for you automatically when you go to reload it. You can then use the GetUserRecord method to return the address of the record variable that was allocated. And you can use GetUserRecordSize to verify that the size of the record variable matches your expectations. This strategy is demonstrated in the entries later in the chapter for EntryScreen.Load and EntryScreen.Store.

2) Alternatively, you can use RegisterPointer to register the address of your user record at the time that you Store it. When you go to reload it, you would call RegisterPointer again to register the address of the new user record. If it is not registered, the Load operation will fail. This strategy is demonstrated in the entries later in the chapter for ScrollingEntryScreen.Load and ScrollingEntryScreen.Store.

What about the case where most of the variables associated with the fields are part of a single record, but not quite all? In this case, you can simply use RegisterPointer to deal with all the exceptions. Consider ENTRY.PAS, for example: all but one of the fields in its main entry screen are associated with the Scrap record variable; CurrentRec is the lone exception. If we wanted to store this entry screen in a stream, we would call SetUserRecord with 'Scrap' and 'SizeOf(Scrap)' as parameters, then register the address of CurrentRec (and, optionally, the address of Scrap).

The ability to store an EntryScreen in a stream and reload it is one of OPENTRY's most powerful features. We know, because it made our task in writing MAKESCRN much easier. But like most powerful tools, it is easily misused. You should not assume, for example, that storing your entry screens on disk and reloading them will automatically make your program smaller. It won't. Although it does allow you to dispense with all the code required to create your entry screens in the first place, it also requires your program to link in all the code required to read them in from a stream. If you have many entry screens in your program, the exchange will probably work to your advantage. But if you only have one or two entry screens, it won't. Our best advice is this: use this feature only if 1) you have many entry screens in your program and/or 2) it allows you to do something that you couldn't easily do any other way.

Data Entry Commands

The editing and field movement commands supported by the EntryScreen are arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command.

Because most of the commands behave differently depending on the field editor in use, the descriptions are generally preceded by one or more abbreviations, indicating the field editor

being described. These are the abbreviations used, followed by the complete name of the field editor, and the corresponding LineEditor methods:

```
SS  SimpleStringEditor  AddSimpleXxxField
SC  SimpleCharEditor    AddSimpleCharField, AddSimpleBooleanField,
                        AddSimpleYesNoField
SE  StringEditor        AddXxxField
CE  CharEditor          AddCharField, AddBooleanField, AddYesNoField
NE  NumberEditor        AddNumericXxxField
MC  ChoiceField.Edit    AddChoiceField
ML  MultiLine.Edit      AddMultiLineField
PE  PickField.Edit      AddPickStringField
```

Now, here are the commands recognized by the various field editors and by the EntryScreen.

Cursor Movement

ccLeft **<Left>, <CtrlS>**
SS, SE, ML: Cursor left one character. If the cursor is already at the beginning of the field, and the efAutoAdvCurBegin option is on, the cursor will be moved left one field.

PE: Cursor left one item.

Else: Cursor left one field.

ccRight **<Right>, <CtrlD>**
SS, SE, ML: Cursor right one character. If the cursor is already at the end of the field, and the efAutoAdvCurEnd option is on, the cursor will be moved right one field.

PE: Cursor right one item.

Else: Cursor right one field.

ccWordLeft **<CtrlLeft>, <CtrlA>**
SS, SE, ML: Cursor left one word.

Else: Cursor left one field.

ccWordRight **<CtrlRight>, <CtrlF>**
SS, SE, ML: Cursor right one word.

Else: Cursor right one field.

ccHome **<Home>, <CtrlQ> <S>**
SS, SE: Cursor to beginning of field.

ML: Cursor to beginning of line.

PE: Cursor to first item in pick list.

ccEnd **<End>, <CtrlQ> <D>**
SS, SE: Cursor to end of field.

ML: Cursor to end of line.

PE: Cursor to last item in pick list.

ccUp **<Up>, <CtrlE>**
ML: If cursor is not on top line of edit field, moves cursor up one line.

PE: Cursor up one item.

ccDown **<Down>, <CtrlX>**
ML: If cursor not on bottom line of edit field, moves cursor down one line.

PE: Cursor down one item.

ccTab **<Tab>, <CtrlI>**
Cursor to next subfield.

ccBackTab **<ShiftTab>**
Cursor to previous subfield.

ccPageUp **<PgUp>**
PE: Cursor up one page (within pick list).

Else: Cursor up one page (within entry screen).

ccPageDn **<PgDn>**
PE: Cursor down one page (within pick list).

Else: Cursor down one page (within entry screen).

ccFirstFld **<CtrlPgUp>, <CtrlQ><R>**
Cursor to first field.

ccLastFld **<CtrlPgDn>, <CtrlQ><C>**
Cursor to last field.

Insertion and Deletion

ccDel **<Del>, <CtrlG>**
SS, SE, NE, ML: Delete character at cursor.

ccBack **<Bksp>, <CtrlH>, <CtrlBksp>**
SS, SE, ML: Delete character to left of cursor.

NE: Delete character at cursor.

ccDelLine **<CtrlY>**
SS, SE, NE: Clear entire field.

ML: Delete current line.

ccDelEol **<CtrlEnd>, <CtrlQ><Y>**
SS, SE: Delete from cursor to end of subfield.

ML: Delete from cursor to end of line.

ccDelBol **<CtrlHome>**
SS, SE: Delete from beginning of subfield to cursor.

ML: Delete from beginning of line to cursor.

ccDelWord **<CtrlT>**
SS, SE: Delete word to right of cursor.

ML: Delete word to right of cursor. If cursor is beyond the end of line, and the following line will fit at the end of the current line, the following line is joined with the current line.

ccCtrlChar **<CtrlP>**
SS, SE, ML: Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is not available if the video attribute for control characters is the same as that for ordinary text and the teMapCtrls option is on.

ccIns **<Ins>, <CtrlV>**
SS, SE, ML: Toggle insert mode on and off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

Other

ccSelect **<Enter>, <CtrlM>, <ClickCenter>**
SS, SC, SE, CE, NE, MC: Accept contents of field.

ML: If in overtype mode, moves cursor to beginning of next line. If in insert mode, inserts new line at position of cursor and moves cursor to beginning of new line (if possible).

PE: Accept current item and advance cursor to the next field. Note that this command will not get the cursor out of the pick list if the field is the last one in the entry screen and the wrap mode is StopAtEdges.

ccDone **<^Enter>, <CtrlJ>, <CtrlK><D>, <CtrlK><Q>**
Accept contents of field.

ccMouseSel **<ClickLeft>**
SS, SC, SE, CE, NE, MC: Move cursor to field pointed to by mouse cursor. If the mouse cursor is on the current field, and the field is a boolean, yes-no, or multiple choice field, the value of the field will be toggled/incremented.

ML, PE: If mouse is within field, moves cursor to position of mouse, else to the field pointed to by the mouse cursor.

ccRestore **<CtrlR>, <CtrlQ><L>**
SS, SC, SE, CE, NE, MC: Restore original contents of the field.

ML: Restore original contents of current line.

ccIncChoice **<+>, <Space>**
MC: Increment a multiple choice field. This command is meaningful only if EditChoice was called. In all other cases, <+> and <Space> will be treated as ordinary characters.

ccDecChoice **<->**
MC: Decrement a multiple choice field. This command is meaningful only if EditChoice was called. In all other cases, <-> will be treated as an ordinary character.

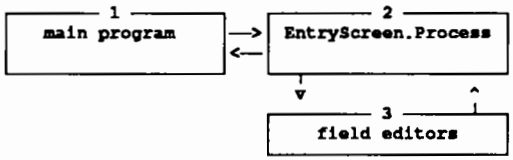
ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established with EntryCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a GetHelpProc, the field editor will pass it the following three parameters: ucEntry, nil, and the value designated as the current field's help index, which defaults to 0.

ccQuit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit editing.

How an EntryScreen Works

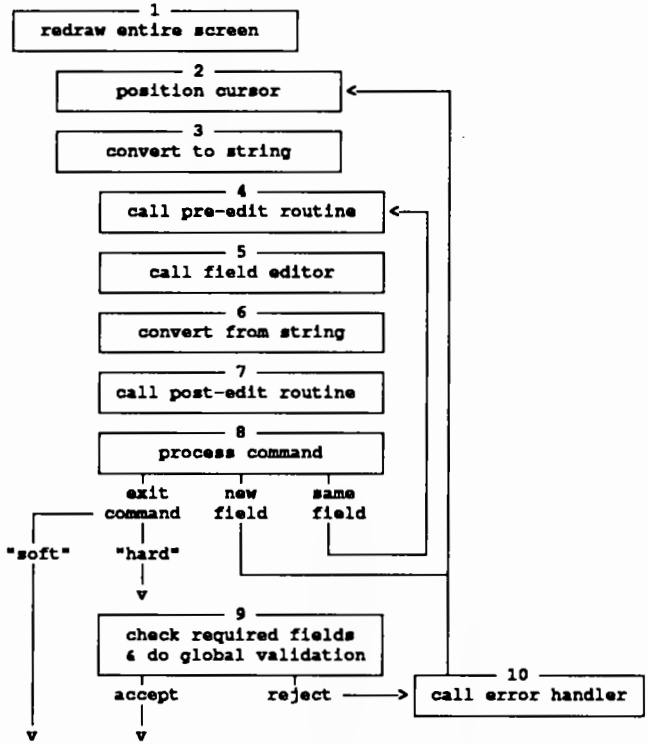
Because entry screens are so complex, we want to step back and give you a broad overview of how the whole data entry system works. Given a general understanding of how control flows from one part of the system to another, you should be better able to understand not only how to use the system, but also how to take advantage of the hooks it makes available to you.

From the broadest view, control flows like this:



That is, the main program calls Process to handle the editing of all the fields in the entry screen. It in turn calls upon one of the *field editors* (see OPFEDIT) to process a particular field. When the field editor returns control, Process will either move the cursor to another field and continue editing, or return control to the caller. Depending on the *exit command* issued by the user, the program may either terminate the editing session or make another call to Process. (For an example of how to process the various exit commands, see ENTRY.PAS.)

Sharpening our focus a bit, we can see that the flow of control within Process itself looks something like this:



On entry to Process, the entire screen is drawn (box 1). (We'll skip the details.) After that, control flows into the larger of the routine's two main loops. At the beginning of this loop (boxes 2-8), the cursor is moved into the current field, then the *conversion routine* for that field is called (see OPFIELD). Its job in this situation is to convert the variable associated with the current field from its native type (Integer, LongInt, Real, Date, etc.) into a string, then merge that string with the field's *picture mask* (see OPABSFLD). This will allow the field editor to perform its duties without any real knowledge of the field's type.

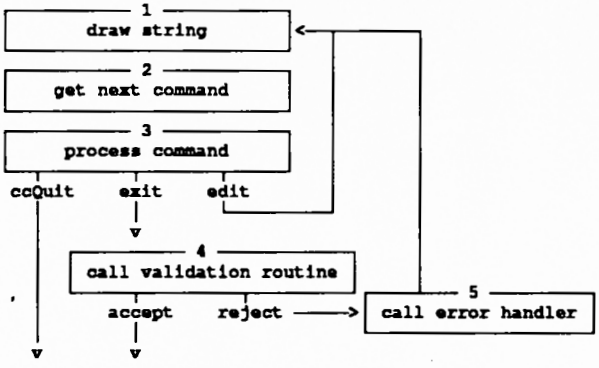
Control then flows into the innermost loop (boxes 4-8). The first task here is to call the *pre-edit routine* (if any). This gives the calling program a chance to display a help message (for example). The next step is to call the field editor. We'll discuss the job of the field editors in more detail below. When the field editor returns control to Process, the conversion routine is called again, this time to convert the string just edited back to its native type.

The next step is to call the *post-edit routine* (if any). This is perhaps the most important hook provided by OPENTRY, and understanding it is the key to mastering the power of the system. You can do many things in a post-edit routine: update a calculated field; protect or unprotect a field; hide or unhide a field; change the command entered by the user; force the cursor to move to a particular field; the list goes on. The more common possibilities are either discussed here in the manual and/or demonstrated in ENTRY.PAS, but the possibilities are almost unlimited.

After the post-edit routine has been called, Process is ready to evaluate the command that caused the field editor to return control. This command will usually fall into one of two categories: an *exit command* (such as <Esc> or <CtrlEnter>) or a cursor movement command (such as <Enter>, <Up>, or <Down>). If it is a cursor movement command, Process has to calculate the field to move to. In some cases, the cursor may not need to be moved at all--for example, if the current wrap mode is StopAtEdges, and the user has tried to move the cursor beyond the last row by pressing <Down>--and control flows back to the top of the inner loop (box 4). In most cases, however, the cursor will be moved to another field, and control will flow back to the top of the outer loop (box 2).

If it is an exit command, Process will do one of two things. If the exit command is "soft," it will return control to the caller. If the exit command is "hard," it will first check all the required fields (if any) and then call the global validation routine (if any). (The "soft" vs. "hard" distinction is explained in the entry for Process.) If either of these two procedures reports an error, the cursor will be positioned on the appropriate field, the error handler will be called, and control will return to the top of the loop. Otherwise it will return control to the caller.

Now let's sharpen our focus a bit further and look at what a typical field editor does:



Most of the field editors have a large main loop. At the top of the loop, the string being edited is drawn and the cursor is moved into position (box 1). The next step is to get a command from the user (box 2) by calling the CommandProcessor associated with the EntryScreen.

Once the keyboard handler has obtained the next keystroke and it has been translated into a command, the field editor must process that command. Here again there are two broad categories of commands: exit commands and editing commands. Editing commands are commands that modify the contents of the field and/or move the cursor from one part of the field another. The category of exit commands comprises both the "true" exit commands (those that are treated as exit commands by Process) and those commands that cause the cursor to be moved from one field to another (otherwise known as *field movement commands*).

When an exit command is given, the field editor will do one of two things. If the command was ccQuit (<Esc>), it will restore the original value of the string being edited and exit immediately. Otherwise, it will call the *validation routine* associated with the current field (see OPFIELD). If the value of the field is acceptable, it will return True, telling the field editor to return control to Process. If it is unacceptable, it will return False, passing back an error code and a pointer to an error message. The field editor will then either call the user-defined error routine if there is one or "ring the bell" if there isn't. (In most applications you will probably want to take advantage of this hook, because the user will rarely understand what he did wrong if you don't display the error message.) This process will be repeated until a valid entry is made by the user or <Esc> is pressed.

Special note to users of Turbo Professional's TPENTRY unit: In TPENTRY, the field editor called the entry screen's pre-edit routine after an error occurred, in case the error message overwrote a help message. This worked well in some cases, but caused awful problems in others. An **Object Professional** field editor never calls the pre-edit routine, so it is the error handler's responsibility to restore whatever portion of the screen it overwrites. See ENTRY.PAS for an example.

Constants

```
BadEntryOptions     : Byte = esFastUpdates+esDeallocUserRec;
```

EntryScreen options that exist for internal use, and may not be altered by calling esOptionsOn or esOptionsOff.

```
BadEsFieldOptions  : LongInt = efMultChoice;
```

Entry field options that exist for internal use, and may not be altered by calling esFieldOptionsOn or esFieldOptionsOff.

```
DefEntryOptions     : Byte = esMousePage;
```

The default options for an EntryScreen or ScrollingEntryScreen.

```
EntryKeyMax = 250;
EntryKeyID  : string[12] = 'opentry keys';
EntryKeySet : array[0..EntryKeyMax] of Byte = (...);
EntryCfgEnd : Byte = 0;
```

EntryKeyId is an ID string used to mark the the beginning of the configuration data area for OPENTRY; EntryCfgEnd marks the end of the configuration data area. In between them is EntryKeySet, the command table used by EntryCommands. EntryKeyMax is the last valid index into the table.

```
esScrollByPage      = $01; {scroll by page?}
esMousePage         = $02; {clicking on scroll bar scrolls by page}
esReadOnly          = $04; {read-only mode}
esDeallocUserRec    = $40; {used internally}
esFastUpdates       = $80; {used internally}
```

Options for an EntryScreen or ScrollingEntryScreen. esScrollbyPage affects the vertical scrolling behavior of a ScrollingEntryScreen when the cursor is moved to a field not currently on screen. If the option is off, the display will be scrolled only enough to reveal the field to move to. If the option is on, the display will be scrolled by a full page (usually equal to the number of rows that can be displayed at one time). esMousePage also applies only to a ScrollingEntryScreen, and it determines what happens when you click on the increment or decrement hot spots of a scroll bar. If the option is off, clicking on these spots is equivalent to pressing <Down> or <Up>, respectively. If the option is on, clicking on them is equivalent to pressing <PgDn> or <PgUp>.

The esReadOnly flag specifies that the EntryScreen should operate in "read-only" mode, meaning that the user may not alter the contents of any of the fields. This option can be useful in database applications in situations where you want to allow the user to see, but not modify, certain database records. The esDeallocUserRec is an internal flag that is sometimes used when an EntryScreen is loaded from a stream. The esFastUpdates option is an internal flag used to speed up screen updates.

Types

```
EntryScreenPtr = ^EntryScreen;
EntryScreen =
  object(AbstractSelector)
    ...
    esLeftD          : Char;       {field delimiters (null = none)}
    esRightD         : Char;
    esPasswordChar   : Char;       {character used in password mode}
    esInsertMode     : Boolean;    {insert mode on?}
    ...
 end;
```

A data entry screen. esLeftD and esRightD are the left and right field delimiters. esPasswordChar is the character used to represent text in password mode. esInsertMode indicates whether the default mode for field editing is insert or overtype.

```
esUserProc = procedure(ESP : EntryScreenPtr);
```

A user-written procedure called when the cursor is about to be moved to a new field (a pre-edit routine) or the cursor is about to leave the current field (a post-edit routine).

```
esUserValidationFunc = function(ESP : EntryScreenPtr;
                                var EFP : EntryFieldPtr;
                                var ErrCode : Word;
                                var ErrMsg : StringPtr) : Boolean;
```

A user-written procedure called to validate the collective contents of a data entry screen.

```
esFieldProc = procedure(EFP : EntryFieldPtr; var D; ESP : EntryScreenPtr);
```

A user-written procedure called by VisitAllEntryFields to perform a particular action on all EntryFields associated with an EntryScreen.

```
{$IFDEF PickListFields}
PickFieldPtr = ^PickField;
PickField =
  object(EntryField)
    pfPick : PickListPtr;   {pointer to the pick list}
  end;
{$ENDIF}
```

The only basic field type implemented in OPENTRY rather than OPFIELD. For a discussion of how to use this field type, see the section on "Pick Fields," above. pfPick is a pointer to the pick list associated with the field.

Variables

```
EntryCommands : CommandProcessor;
```

The default CommandProcessor for an EntryScreen or ScrollingEntryScreen.

ChangeConversion / ChangeRange / ChangeRequired

Declaration
```
procedure ChangeConversion(ID : Word; CP : ConversionProc);
```

Purpose
Change the conversion routine for the specified field.

Description
This method allows you to alter the conversion routine for any field in an entry screen. ID is the ID number for the field. CP is the new conversion routine. You might use ChangeConversion in conjunction with ChangeValidation, for example, to implement a variation on the Word field type that allowed values to be entered and displayed in octal (base 8) notation.

Example
```
ES.ChangeConversion(0, MyConversionProc);
```
Installs MyConversionProc as the conversion routine for field 0.

See Also
ChangeValidation

Declaration
```
procedure ChangeRange(ID : Word; var RangeLo, RangeHi);
```

Purpose
Change the valid range for the specified field.

Description
This method allows you to alter the acceptable range of values for any field in an entry screen. ID is the ID number for the field. RangeLo and RangeHi specify the new lower and upper bounds for the field. The variables passed as RangeLo and RangeHi are assumed to be of the same type as the variable associated with the field.

Example
```
const
  NewLo : Byte = 0;
  NewHi : Byte = 200;
...
ES.AddByteField('Age:', 3, 1, '999', 3, 7, 0, 1, 150, Info.Age);
...
ES.ChangeRange(1, NewLo, NewHi);
```
The field Info.Age is initially set up with a range of 1-150. It is later changed to 0-200 with the call to ChangeRange.

Declaration
```
procedure ChangeRequired(ID : Word; On : Boolean);
```

Purpose
Modify the required status of a field after it has been added.

Description
This method allows you to change the required status for the field specified by ID. If On is True, the field will be marked as required; if On is false, the field will not be required.

Example
```
ES.ChangeRequired(0, True);
```
Make the first field in the entry screen a required field.

Declaration
```
procedure ChangeValidation(ID : Word; VF : ValidationFunc);
```

Purpose
Change the validation routine for the specified field.

Description
This method allows you to alter the validation routine for any field in an entry screen. ID is the ID number for the field. VF is the new validation routine. You might use ChangeValidation to specify a special validation routine for a field of type String, as shown in the example below. Or you might want to use two different validation routines for a given field, with the choice of routines depending on whether validation needs to be strict or relaxed in a given situation.

Example
```
ES.ChangeValidation(0, ValidateNoBlanks);
```
Requires the first field in the ES object to be completely full (contain no blanks).

See Also
ChangeConversion

Declaration
```
function CurrentFieldModified : Boolean;
```

Purpose
Return True if current field was modified.

Description
This method is intended to be used within a post-edit routine (only!) to determine whether the current field was actually modified by the user. It is especially useful in applications where the post-edit routine must load data from disk based on the value of a particular field, and you don't want to reread the same data unnecessarily.

Note that there is one case where CurrentFieldModified is unreliable. If the user executes the ccMouseSel command (normally assigned to <ClickLeft>) while the mouse cursor is on a boolean, yes-no, or multiple choice field, and that field is the current field, the value of that field will ultimately be toggled or incremented. But the change in value will not occur until *after* the post-edit routine has been called. You can detect this case, however, using EvaluateCommand. See the entry for that method for details.

Example
See the example for EvaluateCommand.

See Also
EvaluateCommand

Done / esFieldOptionsAreOn / esFieldOptionsOff

Declaration
```
destructor Done; virtual;
```

Purpose
Dispose of the entry screen.

Description
This destructor disposes of the user record, if one was allocated dynamically by Load, then calls the parent's destructor (AbstractSelector.Done) to dispose of the field list, screen buffers, etc.

Example
```
ES.Erase;
ES.Done;
```
Erase ES and release all heap space.

See Also
AbstractSelector.Done

Declaration
```
function esFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified primary field options are on.

Description
This method returns True if the specified primary field option(s) are currently selected.

Example
```
if ES.esFieldOptionsAreOn(sfMapCtrls) then
  ES.esFieldOptionsOff(efMapCtrls)
else
  ES.esFieldOptionsOn(efMapCtrls);
```
Toggle the state of the efMapCtrls option.

See Also
esFieldOptionsOn

Declaration
```
procedure esFieldOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple field options.

Description
This method deactivates the specified primary field option(s), excluding those designated as BadEsFieldOptions.

Example
```
ES.esFieldOptionsOn(efProtected);
ES.AddRealField('Weekly:',4,1, '99999.99', 4,9,0, 0.0, 0.0, 2,Info.Weekly);
ES.esFieldOptionsOff(efProtected);
```
The field for Info.Weekly will be a protected field.

See Also
esFieldOptionsOn

esFieldOptionsOn / esOptionsAreOn / esOptionsOff

Declaration

```
procedure esFieldOptionsOn(OptionFlags : LongInt);
```

Purpose

Activate multiple field options.

Description

This method activates the specified primary field option(s), excluding those designated as BadEsFieldOptions.

Example

```
ES.esFieldOptionsOn(efProtected);
ES.AddRealField('Weekly:', 4, 1, '99999.99',
                4, 9, 0, 0.0, 0.0, 2, Info.Weekly);
ES.esFieldOptionsOff(efProtected);
```

The field for Info.Weekly will be a protected field.

See Also

esFieldOptionsAreOn esFieldOptionsOff

Declaration

```
function esOptionsAreOn(OptionFlags : Byte) : Boolean;
```

Purpose

Return True if all specified options are on.

Description

This method returns True if the specified EntryScreen option(s) are currently selected.

Example

```
if ES.esOptionsAreOn(slScrollByPage) then
  ES.esOptionsOff(esScrollByPage)
else
  ES.esOptionsOn(esScrollByPage);
```

Toggles option to scroll by page.

See Also

esOptionsOn

Declaration

```
procedure esOptionsOff(OptionFlags : Byte);
```

Purpose

Deactivate multiple options.

Description

This method deactivates the specified EntryScreen option(s), excluding those designated as BadEntryOptions (see the Constants section, above).

Example

```
ES.esOptionsOff(esReadOnly);
```

Returns the entry screen to input mode.

See Also

esOptionsOn

esOptionsOn / esSecFieldOptionsAreOn

Declaration

```
procedure esOptionsOn(OptionFlags : Byte);
```

Purpose

Activate multiple options.

Description

This method activates the specified EntryScreen option(s), excluding those designated as BadEntryOptions (see the Constants section, above).

Example

```
ES.esOptionsOn(esReadOnly);
```

Puts the data entry screen in read-only mode.

See Also

esOptionsAreOn esOptionsOff

Declaration

```
function esSecFieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose

Return True if all specified secondary field options are on.

Description

This method returns True if the specified secondary field option(s) are currently selected.

Example

```
if ES.esSecFieldOptionsAreOn(sefPadCurrentOnly) then
  ES.esSecFieldOptionsOff(sefPadCurrentOnly)
else
  ES.esSecFieldOptionsOn(sefPadCurrentOnly);
```

Toggle the state of the sefPadCurrentOnly option.

See Also

esSecFieldOptionsOff esSecFieldOptionsOn

Declaration
```
procedure esSecFieldOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple secondary field options.

Description
This method deactivates the specified secondary field option(s), excluding those designated in OPFIELD as BadSEFieldOptions (see the documentation for OPFIELD).

Example
```
ES.esSecFieldOptionsOff(sefSuppressZero);
```
Turn off option to suppress display of 0's in integer type fields.

See Also
esSecFieldOptionsAreOn

Declaration
```
procedure esSecFieldOptionsOn(OptionFlags : LongInt);
```

Purpose
Activate multiple secondary field options.

Description
This method activates the specified secondary field option(s), excluding those designated in OPFIELD as BadSEFieldOptions (see the documentation for OPFIELD).

Example
```
ES.SetPadChar('_');
ES.esSecFieldOptionsOn(sefPadCurrentOnly);
```
Pad fields with underscores only when they are being edited.

See Also
esSecFieldOptionsAreOn

EvaluateCommand

Declaration

```
function EvaluateCommand(var Cmd : Word) : Word;
```

Purpose

Given a command, return the ID for the field the cursor will move to next.

Description

This method is intended to be called only from within a post-edit routine. Given the code for a command to be evaluated (usually the last command entered by the user, returned by GetLastCommand), it calculates where the EntryScreen will be moving the cursor after the post-edit routine returns control. If the cursor is not going to be moved at all, the function result will return the ID for the current field; otherwise, it returns the ID for the field to move to.

EvaluateCommand is especially useful in cases where a post-edit routine needs to load additional data from disk after the user has entered a value in a particular field. For example, after the user enters a customer ID number, the post-edit routine might need to go to disk to load the rest of the known data concerning that customer (name, address, etc.).

The reason Cmd is a var parameter is that in some cases it will be modified upon return. For example: if the cursor is on the bottom row of the entry screen, the wrap mode is ExitAtEdges, and the initial value for Cmd is ccDown, then the value of Cmd upon return will be ccExitAtBot. (The function result in that case will contain the ID for the current field, since the cursor will not be moved to another field.)

A perhaps more important case is the one described in the entry for CurrentFieldModified, where the user has executed the ccMouseSel command on a boolean, yes-no, or multiple choice field when that field is current. In such a case, EvaluateCommand can be used to predict whether or not the field *will be* modified as a result of the user's command. If the initial value for Cmd is ccMouseSel, and the value upon return is ccIncChoice, then you know that the command will result in the field's value being modified.

Example

```
{$F+}
procedure PostEdit(ESP : EntryScreenPtr);
var
  Cmd : Word;
begin
  if (ESP^.GetCurrentID = 0) and ESP^.CurrentFieldModified then begin
    Cmd := ESP^.GetLastCommand;
    if ESP^.EvaluateCommand(Cmd) <> 0 then begin
      {*** cursor is being moved--load new data from disk ***}
      {force entire entry screen to be redrawn}
      ESP^.ResetScreen;
    end;
  end;
end;
{$F-}
```

This example assumes that if field 0 is modified, and the cursor is being moved to another field, we need to load some data from disk and redraw the entire entry screen.

See Also

CurrentFieldModified

Declaration
```
procedure GetFieldCoordinates(var Row, Col, Wid : Byte);
```

Purpose
Returns the absolute screen coordinates for the current field.

Description
This method is intended to be called only from within a pre-edit routine. What it returns are the absolute screen coordinates for the soon-to-be current field (Row and Col), along with its width (Wid).

Probably the most common application for it is in a case where a scrolling entry screen contains a pick field, and the pick list needs to be positioned such that it does not obscure the user's view of the field itself. Based on the information provided by GetFieldCoordinates, the pre-edit routine can move the associated pick list to an appropriate location on the screen. See the discussion of "Pick Fields," above, for an explanation of why you might want to do this.

Example
```
ES.GetFieldCoordinates(Row, Col, Wid);
```
Returns the coordinates of the current field.

Declaration
```
function GetLastField : EntryFieldPtr;
```

Purpose
Return pointer to last field in entry screen.

Description
This method returns a pointer to the last field added to the entry screen. You might use this value to change the options for that field, as shown in the example below, but it is better used in cases where the address of a particular field needs to be used frequently later in the program. By calling GetLastField immediately after adding the field and saving the address in a global variable, you can save time later by avoiding the need to call FindField in order to locate it. (FindField, a method in AbstractSelector that locates a field based on its ID number, sometimes has to scan half the fields in the list in order to find a particular one.)

Important: Before using the pointer returned by GetLastField, you must be sure that it is not nil. If no fields have yet been added to the EntryScreen (presumably because of an as-yet undetected error), GetLastField will return nil.

Example
```
var
  EFP : EntryFieldPtr;
...
  EFP := ES.GetLastField;
  if EFP <> nil then
    EFP^.efOptionsOn(efRightJustify);
```
Set the efRightJustify option for the last field added to the entry screen.

GetUserRecord / GetUserRecordSize

Declaration
```
function GetUserRecord : Pointer;
```

Purpose
Return a pointer to the user record.

Description
This method returns a pointer to the "user record." It is needed only in cases where an EntryScreen has been loaded from a stream and the Load constructor has had to allocate the record used to store data being edited. See the example for the Load method and the discussion of "EntryScreens and Streams" in the introduction to this section.

Example
See the example for Load.

See Also
Load

Declaration
```
function GetUserRecordSize : Word;
```

Purpose
Return the size of the user record.

Description
This method returns the size of the "user record." It is needed only in cases where an EntryScreen has been loaded from a stream and the Load constructor has had to allocate the record used to store data being edited. See the example for the Load method and the discussion of "EntryScreens and Streams" in the introduction to this section.

Example
See the example for Load.

See Also
Load

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Initialize the entry screen.

Description

This constructor instantiates an EntryScreen using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not ES.Init(25, 5, 55, 11) then begin
  WriteLn('Failed to init EntryScreen. Status = ', InitStatus);
  Halt;
end;
```

Instantiates an unbordered EntryScreen window using default colors.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
```

Purpose

Initialize the entry screen with custom window options.

Description

This constructor instantiates a EntryScreen using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window.

InitCustom will fail only if the parent constructor (AbstractSelector.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Several sets of attributes in the specified ColorSet have special relevance for window objects of type EntryScreen (or ScrollingEntryScreen). Most of these are relevant for all descendants of the AbstractSelector object (see the entry for AbstractSelector.InitCustom). But there are three other pairs of attributes in the ColorSet passed to InitCustom that are relevant only for EntryScreens, all of them pertaining to field delimiters: DelimColor/Mono, SelDelimColor/Mono, and ProDelimColor/Mono. After the EntryScreen has been instantiated, these attributes can be changed by calling SetDelimAttr, SetSelectedDelimAttr, and SetProtectedDelimAttr, respectively.

Example

```
if not ES.InitCustom(25, 5, 55, 11,                {Window coordinates}
                     DefaultColorSet,              {ColorSet}
                     DefWindowOptions or wBordered) {Window options}
then begin
  WriteLn('Failed to Init EntryScreen,  Status = ', InitStatus);
  Halt;
end;
```

Instantiates a bordered EntryScreen window using default colors.

See Also

AbstractSelector.InitCustom Init

Load

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load an entry screen from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to EntryScreen.Store.

The various field types used by the EntryScreen must be registered before calling Load, as shown in the example below. If appropriate, the address of the user record should be registered as well, and so should the addresses of any user-written routines associated with the EntryScreen (pre-edit, post-edit, global validation, increment choice).

The stream registration procedure for an EntryScreen is EntryScreenStream.

For further details, see the discussion of "EntryScreens and Streams" in the introduction to this section.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpField, OpSelect, OpEntry;
type
  InfoRec = record
              Name : string[30];
              Age : Byte;
              Wage : Real;
            end;
  InfoRecPtr = ^InfoRec;
var
  InfoPtr : InfoRecPtr;
  ES : EntryScreen;
  S : BufIdStream;
begin
  S.Init('ENTRY.STM', SOpen, 1024);
  S.RegisterHier(EntryScreenStream);
  S.RegisterHier(StringFieldSimStream);
  S.RegisterHier(ByteFieldStream);
  S.RegisterHier(RealFieldStream);
  S.Get(ES);
  if S.GetStatus <> 0 then
    Halt;
  S.Done;
  if SizeOf(InfoRec) <> ES.GetUserRecordSize then begin
    WriteLn('Stream mismatch');
    Halt;
  end;

  InfoPtr := ES.GetUserRecord;
  FillChar(InfoPtr^, SizeOf(InfoRec), 0);

  ClrScr;
  ES.Process;
  ES.Erase;
  ES.Done;
end.
```

Instantiates ES from a stream created by the Store example using the stream's Get method, which calls EntryScreen.Load indirectly. Notice that, in addition to registering the EntryScreen type, we must also register all of the field types used by it.

Since the Store example didn't write out a pointer to the user record, Load automatically allocates heap space for it. It's therefore up to our application to retrieve a pointer to the user record by calling GetUserRecord, as well as to initialize the user record (done here with FillChar).

See Also

Store

Declaration

```
procedure Process; virtual;
```

Purpose

Process data entry commands.

Description

EntryScreen's Process method follows the general model described under CommandWindow.Process with the following additions and exceptions:

It handles the commands described in "Data Entry Commands" at the start of this section.

If the field designated as the current field is protected, Process will move the cursor to the first unprotected field it can find. If there are no unprotected fields, Process exits with the error epFatal+ecNoFields.

Process makes direct calls to up to 3 user-written routines. Before moving the highlight to a new field, Process calls the pre-edit routine. Before leaving the current field, it calls the post-edit routine. And before allowing the user to accept the entry screen, it calls the global validation routine. See the discussion of these hooks at the beginning of this section for details.

The exit commands recognized by Process fall into two categories, "soft" and "hard." When a user issues one of the following (soft) exit commands, Process assumes that it does not need to check to see if all the required fields have been filled in, and that it does not need to call the global validation routine:

```
ccQuit, ccUser0..ccUser65335, ccClickExit, ccNested
```

The reasoning here is that 1) ccQuit means "ignore all changes and quit editing," and 2) the other commands are generally or frequently used to exit from the entry screen only temporarily, hence the characterization of these commands as "soft."

When the user issues a command in the second group, however, Process does check the required fields and it does call the global validation routine:

```
ccDone, ccNextRec, ccPrevRec, ccExitAtTop, ccExitAtBot, *ccMouseSel
```

These exit commands are ones that are likely to lead to the EntryScreen being destroyed or having new data loaded into it–hence the characterization of these commands as "hard."

* Note that ccMouseSel is treated as an exit command only if the wAllMouseEvents option is in effect, and the ccMouseSel command was generated while the mouse was on or outside the window frame. If the mouse was inside the window or on a scroll bar, Process will handle the mouse event itself.

Example

See example at the beginning of this section.

RevalidateAll / SetBeepOnError

Declaration

```
function RevalidateAll(var EFP : EntryFieldPtr; var ErrCode : Word;
                       var ErrMsg : StringPtr) : Boolean;
```

Purpose

Re-validate all unprotected fields in entry screen.

Description

This method, which is intended to be called only from a global validation routine, simply invokes the validation routine for each field in the entry screen. If one of the validation routines reports an error, RevalidateAll will return False, EFP will point to the field containing the error, ErrCode will contain the code for the error, and ErrMsg will point to a string that describes the error.

RevalidateAll exists because, if the validation routine for a given field was executed only after the field had been edited, and the user exited from the entry screen without editing all the fields, some of its values would not be validated. In most applications, this is not a problem. But if the default values for some of the fields were invalid–e.g., if the default value for a Date field were BadDate, but the acceptable range of values for the field was 1/1/1980-1/1/2000–then it would be possible for the user to exit from the entry screen while it still contained an invalid entry.

With TPENTRY, the forerunner of OPENTRY, this scenario was a real possibility. In OPENTRY, however, the default global validation routine automatically calls RevalidateAll for you. The method is documented here in case you need to use a customized global validation routine instead of the default one.

Example

See the example for SetValidateFunc.

See Also

SetValidateFunc

Declaration

```
procedure SetBeepOnError(On : Boolean);
```

Purpose

Activate/deactivate the beep-on-error option for all fields.

Description

This method activates (On = True) or deactivates (On = False) the efBeepOnError option for all fields in an entry screen. For a description of this option, see the section on primary EntryField options in the documentation for OPFIELD.

Example

```
ES.SetBeepOnError(True);
```

Turns on beep-on-error option for all fields.

Declaration

```
procedure SetDelimAttr(Color, Mono : Byte);
```

Purpose

Set attributes for delimiters (unselected).

Description

This method allows you to specify the attributes in which the delimiters for unselected fields are displayed on Color and Monochrome systems. Note that the new attributes affect all fields in the entry screen.

Example

```
ES.SetDelimAttr($02, $70);
```

Delimiters for unselected fields, if used, will be displayed in green on black for color monitors, reverse video for monochrome monitors.

See Also

SetProtectedDelimAttr SetSelectedDelimAttr

Declaration

```
procedure SetDelimiters(Left, Right : Char);
```

Purpose

Set field delimiters for the entry screen (null = none).

Description

This method allows you to specify a pair of characters that will be displayed immediately to the left and right of each field in the entry screen. Note that neither delimiter will be displayed if either left or right is a null character (#0). For further details, see the discussion of "Field Delimiters," above.

Example

```
ES.SetDelimiters('[', ']');
```

Delimit all fields with square brackets.

See Also

SetProtectedDelimAttr SetSelectedDelimAttr

SetForceMode / SetPadChar

Declaration

```
procedure SetForceMode(Force, Overtype : Boolean);
```

Purpose

Force insert or overtype mode, else use previous setting.

Description

This method allows you to indicate that certain fields may be edited only in insert or overtype mode. If Force is True, all fields added after the call to SetForceMode will be edited in either insert (Overtype = False) or overtype (Overtype = True) mode. If SetForceMode is called a second time with a Force parameter of False, fields added subsequently will be edited normally (i.e., the insert/overtype mode setting will be retained from one field to the next). The Overtype parameter is meaningless if Force is False.

Note that SetForceMode does not do anything that could not be done with esFieldOptionsOn and esFieldOptionsOff. The following method calls are equivalent:

```
SetForceMode(True, True);   esOptionsOn(efForceMode+efForceOvertype);
SetForceMode(True, False);  esOptionsOn(efForceMode);
                            esOptionsOff(efForceOvertype);
SetForceMode(False, xxx);   esOptionsOff(efForceMode);
```

Note too that the forced mode setting has no effect on fields of type Char, Boolean, or YesNo, nor does it affect numeric fields (fields that use NumberEditor).

Example

```
ES.SetForceMode(True, False);
ES.AddSimpleStringField('Name:', 2, 1, 'A', 2, 9, 10, 20, 0, Info.Name);
ES.SetForceMode(False, False);
```

Forces the current field back to insert mode whenever it becomes the current field. After the field has been edited, the old mode is restored.

See Also

esFieldOptionsOff esFieldOptionsOn

Declaration

```
procedure SetPadChar(Ch : Char);
```

Purpose

Set character used to pad ends of strings.

Description

This method allows you to specify the character used to pad the end of a field being edited. For example, if you had a field 20 characters wide and a pad character of '_', the field would look something like this on the screen:

```
Name: John Smith__________
```

The default pad character is DefPadChar (= ' '). Unlike SetPasswordChar, SetPadChar affects only those fields added to the entry screen after the call (theoretically, each field in the entry screen could have a different pad character).

Note that the pad character affects only the way that the field is displayed. It does not affect the actual contents of the field.

Example

```
ES.SetPadChar('_');
ES.AddSimpleStringField('Name:', 2, 1, 'A', 2, 9, 10, 20, 0, Info.Name);
ES.SetPadChar(' ');
```

Sets the pad character for this field to '_'.

Declaration
```
procedure SetPasswordChar(Ch : Char);
```

Purpose
Set character used in password mode.

Description
This method allows you to specify the character used to represent ordinary text within a field while in password mode. For example, suppose we have a picture mask of '999-999-9999' (for a phone number field), a pad character of '_', and a password character of '*'. Here's what the field would look like in both normal and password modes at different stages of the entry process:

```
Normal:    ___-___-____   415-55_-____   415-555-12__   415-555-1212
Password:  ___-___-____   ***-**_-____   ***-***-**__   ***-***-****
```

The default password character is DefPasswordChar (= ^G). Note that this setting applies to all fields in the entry screen. For further details concerning password mode, see the discussion of "Field Drawing Routines" in the OPFIELD section.

Example
```
ES.SetPasswordChar('*');
ES.esFieldOptionsOn(efPasswordMode);
ES.AddSimpleStringField('Password:', 2, 1, 'A', 2, 10, 10, 20, 0, PW);
ES.esFieldOptionsOff(efPasswordMode);
```

Sets the password character to '*' and indicates that the field added should be edited in password mode.

Declaration
```
procedure SetPostEditProc(PEP : esUserProc);
```

Purpose
Specify routine to call after each edit.

Description
This method allows you to specify a post-edit routine (PEP), which will be called by Process after the user has edited the current field.

A post-edit routine must be of the following form:

```
{$F+}
procedure PostEdit(ESP : EntryScreenPtr);
begin
end;
```

The ESP parameter is a pointer to the EntryScreen making the call (@Self is passed by Process). See the discussion of "Post-Edit Routines and Calculated Fields" at the beginning of this section for further details.

Example
See the example program at the beginning of this section.

See Also
SetPreEditProc

SetPreEditProc / SetProtectedDelimAttr

Declaration

```
procedure SetPreEditProc(PEP : esUserProc);
```

Purpose

Specify routine to call just before field is edited.

Description

This method allows you to specify a pre-edit routine (PEP), which will be called by Process just before the field editor is called to edit the current field.

A pre-edit routine must be of the following form:

```
{$F+}
procedure PreEdit(ESP : EntryScreenPtr);
begin
end;
```

The ESP parameter is a pointer to the EntryScreen making the call (@Self is passed by Process). See the discussion of "Pre-Edit Routines" at the beginning of this section for further details.

Example

See the example program at the beginning of this section.

See Also

SetPostEditProc

Declaration

```
procedure SetProtectedDelimAttr(Color, Mono : Byte);
```

Purpose

Set attributes for delimiters (protected).

Description

This method allows you to specify the attributes in which the delimiters for protected fields are displayed on Color and Monochrome systems. Note that the new attributes affect all fields in the entry screen.

Example

```
ES.SetProtectedDelimAttr($06, $70);
```

If delimiters are in use, the delimiters for protected fields will be displayed in brown on black for color monitors, reverse video for monochrome monitors.

See Also

SetDelimAttr SetSelectedDelimAttr

Declaration

```
procedure SetSelectedDelimAttr(Color, Mono : Byte);
```

Purpose

Set attributes for delimiters (selected).

Description

This method allows you to specify the attributes in which the delimiters for selected fields are displayed on Color and Monochrome systems. Note that the new attributes affect all fields in the entry screen.

Example

```
ES.SetSelectedDelimAttr($03, $70);
```

Delimiters for the selected field, if used, will be displayed in cyan on black for color monitors, reverse video for monochrome monitors.

See Also

SetDelimAttr SetProtectedDelimAttr

Declaration

```
procedure SetUserRecord(var UserRec; UserRecSize : Word);
```

Purpose

Set the address and size of the user record.

Description

This method allows you to specify the address and size of a "user record." It is needed only in cases where an EntryScreen is to be stored in a stream. See the example for the Store method and the discussion of "EntryScreens and Streams" in the introduction to this section.

Example

See the example for Store.

See Also

Store

SetValidateFunc

Declaration

```
procedure SetValidateFunc(UVF : esUserValidationFunc);
```

Purpose

Specify routine to call before exiting.

Description

This method allows you to specify a global validation routine (UVF) that will be called by Process just before it returns permanent control to the caller. Its job is to examine the contents of all fields in the entry screen to determine whether its collective contents are acceptable.

A global validation routine must be of the following form:

```
{$F+}
function ValidateFunc(ESP : EntryScreenPtr; var EFP : EntryFieldPtr;
                      var ErrCode : Word; var ErrorSt : StringPtr) : Boolean;
begin
end;
```

ESP is a pointer to the entry screen to be validated. If a problem is found, the function must return False and set the three var parameters as follows: EFP must point to the field that caused the error, ErrCode should contain the code for the error, and ErrorSt must point to a message that describes the error.

For further details, see the discussion of "Global Validation Routines" at the beginning of this section.

Example

```
{$F+}
function MyValidateFunc(ESP : EntryScreenPtr; var EFP : EntryFieldPtr;
                        var ErrCode : Word; var ErrorSt : StringPtr) : Boolean;
begin
  {** check for specific problems **}
  MyValidateFunc := ESP^.RevalidateAll(EFP, ErrCode, ErrMsg);
end;
{$F-}
...
ES.SetValidateFunc(MyValidateFunc);
```

A simple global validation routine that uses RevalidateAll.

See Also

Process RevalidateAll

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store an entry screen in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the EntryScreen to the stream S. The various field types used by the EntryScreen must be registered before calling Store, as shown in the example below.

If appropriate, the address of the user record should be registered as well, and so should the addresses of any user-written routines associated with the EntryScreen (pre-edit, post-edit, global validation, IncChoice).

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for an EntryScreen is EntryScreenStream.

For further details, see the discussion of "EntryScreens and Streams" in the introduction to this section.

Example

```
uses OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpField, OpSelect, OpEntry;
type
  InfoRec = record
              Name : string[30];
              Age : Byte;
              Wage : Real;
            end;
var
  Info : InfoRec;
  ES : EntryScreen;
  S : BufIdStream;
  Status : Word;
begin
  if not ES.InitCustom(25, 5, 55, 11, DefaultColorSet,
                       DefWindowOptions or wBordered) then
    Halt;
  ES.AddSimpleStringField('Name:', 2, 1, 'A', 2, 7, 10, 20, 0, Info.Name);
  ES.AddByteField('Age:', 3, 1, '999', 3, 7, 0, 1, 150, Info.Age);
  ES.AddRealField('Wage:',4, 1, '999.99', 4, 7, 0, 0.0, 0.0, 2, Info.Wage);
  {Store this entry screen}
  ES.SetUserRecord(Info, SizeOf(Info));
  S.Init('ENTRY.STM', SCreate, 1024);
  S.RegisterHier(EntryScreenStream);
  S.RegisterHier(StringFieldSimStream);
  S.RegisterHier(ByteFieldStream);
  S.RegisterHier(RealFieldStream);
  S.Put(ES);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  ES.Done;
end.
```

Store

Instantiates ES, adds fields to it, and then stores ES in a stream using the stream's Put method, which calls Store indirectly. We call SetUserRecord to tell ES about the location and size of our data record, to allow it to be "reconstructed" when the EntryScreen is reloaded.

Note that, in addition to registering the EntryScreen type, we've also had to register each of the field types used by it.

See Load for an example program that reloads this EntryScreen.

See Also

Load

Declaration

```
procedure VisitAllEntryFields(FProc : esFieldProc; var D);
```

Purpose

Call the specified procedure for all fields in the entry screen.

Description

This method allows you to specify a user-written procedure to be called for each EntryField associated with an EntryScreen. FProc is the procedure to be called, and D is the data to be passed to it.

FProc must be a procedure of the form:

```
{$F+}
procedure FieldProc(EFP : EntryFieldPtr; var D; ESP : EntryScreenPtr);
begin
end;
```

EFP is a pointer to the EntryField to be modified. D is the data passed to VisitAllEntryFields. And ESP is a pointer to the EntryScreen itself.

VisitAllEntryFields is a low-level routine and must be used with caution. It is always safe to change video attributes, as shown in the example below. But global changes to option settings should generally be avoided. Activating the efTrimBlanks option for every field, for example, can lead to serious problems with certain types of fields (notably array of char fields).

Example

```
const
  BothAttr : Word = $0B70;

  {$F+}
  procedure GlobalPromptAttr(EFP : EntryFieldPtr; var D; ESP : EntryScreenPtr);
  var
    W : Word absolute D;
  begin
    EFP^.SetPromptAttr(Hi(W), Lo(W));
  end;
  {$F-}
...
ES.VisitAllEntryFields(GlobalPromptAttr, BothAttr);
```

Changes the unselected prompt attribute of for all selector fields by "visiting" each field with the procedure GlobalPromptAttr.

See Also

AbstractSelector.VisitAllSelectFields AbstractSelector.VisitAllTextFields

2. The ScrollingEntryScreen

As we indicated in the introduction to the OPENTRY section, the EntryScreen object just described has two important limitations: 1) all the fields associated with it must fit within the window at one time, and 2) the window cannot be zoomed or resized. The purpose of the ScrollingEntryScreen is to provide an EntryScreen free of these limitations.

Compare the following program to the one at the beginning of the previous section:

```
{$V-}                          {<- required for OPENTRY}
program ScrollingEntryExample; {EXENTRY4.PAS}
uses
  OpString, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEntry;
type
  InfoRec = record
              Name : String[30];
              Age : Byte;
              Wage : Real;
              Misc : array[1..10] of String[10];
            end;
var
  Info : InfoRec;
  ExitCommand : Word;
  SES : ScrollingEntryScreen;
  Finished : Boolean;
  I : Word;
begin
  ClrScr;
  FillChar(Info, SizeOf(Info), 0);
  if not SES.InitCustom(25, 5, 55, 10,                {Window coordinates}
                        DefaultColorSet,              {ColorSet}
                        DefWindowOptions OR wBordered) {Window options}
  then begin
    WriteLn('Failed to init EntryScreen. Status = ', InitStatus); Halt;
  end;

  {add window frills}
  SES.wFrame.AddShadow(shBR, shOverWrite);
  SES.wFrame.AddHeader(' Scrolling Entry Screen ', heTC);

  {set wrap mode}
  SES.SetWrapMode(StopAtEdges);

  {Add our data entry fields}
  SES.esFieldOptionsOn(efClearFirstChar);
  SES.AddStringField('Name:', 2, 1, '', 2, 7, 20, 0, Info.Name);
  SES.AddByteField('Age:', 3, 1, '999', 3, 7, 0, 1, 150, Info.Age);
  SES.AddRealField('Wage:',4, 1, '999.99', 4, 7, 0, 0.0, 0.0, 2, Info.Wage);
  for I := 1 to 10 do
    SES.AddStringField(
      'Misc'+Long2Str(I), I+4, 1, '', I+4, 7, 10, 0, Info.Misc[I]);

  {allocate the virtual screen}
  SES.AllocateScreen;

  {check for errors}
  I := SES.GetLastError;
  if I <> 0 then begin
    WriteLn('Error ', I, ' creating entry screen'); Halt;
  end;
```

```
  Finished := False;
  repeat
    SES.Process;
    case SES.GetLastCommand of
      ccQuit, ccDone : Finished := True;
      ccError :
        begin
          WriteLn('Fatal error ', SES.GetLastError);
          Finished := True;
        end;
      {... process other exit commands ...}
    end;
  until Finished;

  SES.Erase;
  SES.Done;
end.
```

Apart from the absence of pre-edit and post-edit routines, the only real differences between this program and the earlier one are 1) this one has more fields (the calculated field Weekly is gone and 10 new string fields have been added), 2) this one has a call to AllocateScreen after all of the fields have been added, and 3) GetLastError is called after AllocateScreen. The most important of these is the second.

What AllocateScreen does is allocate a virtual screen on which all of the fields in the list can be drawn at once. Once the entire virtual screen has been filled in, the ScrollingEntryScreen will determine which section of the virtual screen needs to be copied to the physical screen in order to display the current field, then copy it. As the cursor is moved from field to field, the ScrollingEntryScreen will check to see whether or not the new field is already being displayed on the screen. If it is, only that field needs to be redrawn. If not, it will quickly copy a different section of the (already drawn) virtual screen to the physical screen, then redraw the current field. Although the same effect could be achieved without use of a virtual screen, screen updates would be significantly slower.

Because virtual screens can be very large--nearly 64K--it is possible to have literally hundreds of fields associated with a single ScrollingEntryScreen (400 is a good upper limit). In the case of our example program, we have 13 fields displayed within a window that is 6 rows high. If you run the program and press the <Down> key repeatedly, you'll see the first 6 fields progressively scroll out of view once you reach the bottom of the window, and the last 7 fields will scroll into view.

The ScrollingEntryScreen depends so heavily on its virtual screen that it cannot even draw itself once without it. This is the reason for the call to GetLastError in the example program. If AllocateScreen failed to allocate the virtual screen, there's no point in going any farther. Although it will be called for you automatically if you fail to do it, the error condition will generally be easier to handle if you make the call yourself.

Even if all your fields fit into an EntryScreen's window, you may want to use a ScrollingEntryScreen anyway, because it can be zoomed and resized. This is a by-product of the technique that ScrollingEntryScreen uses to update the window. Since it cannot assume, as EntryScreen must, that all the fields will fit in the window, the size of the window doesn't really matter as long as the largest field in the list will fit. (ScrollingEntryScreen insures that it will by setting a minimum size for the window.)

Important: Please be sure to read the discussion of the ResetScreen method in the introduction to the AbstractSelector section before using a ScrollingEntryScreen.

Types

```
ScrollingEntryScreen =
  object(EntryScreen)
    ...
    sesRowOfs      : Integer;      {number of rows above top of window}
    sesColOfs      : Integer;      {number of cols to left of window}
    ...
  end;
```

A scrollable version of the EntryScreen object. The only data fields of general interest in a ScrollingEntryScreen are sesRowOfs and sesColOfs. sesRowOfs keeps track of the number of rows that have been scrolled out of view: if row 8 was being displayed at the top of the window, for example, sesRowOfs would be 7. sesColOfs is the number of columns that have been scrolled out of view: if column 25 was being displayed at the left edge of the window, ssColOfs would be 24.

Declaration
```
procedure AllocateScreen;
```

Purpose
Call after last field added to allocate virtual screen.

Description
This method should be called after the last field has been added to the ScrollingEntryScreen. It scans the list of fields (including text fields) to determine the maximum row and column values for any field in the list, then tries to allocate a virtual screen large enough to allow all the fields to be drawn on it at once. To determine whether or not the virtual screen was allocated successfully, you should call GetLastError immediately following the call to AllocateScreen. Assuming that there was no error pending prior to the call, the only error code GetLastError should return is epFatal+ecOutOfMemory.

Note: If you neglect to call AllocateScreen before calling Draw or Process, AllocateScreen will be called for you. In most cases, however, it is easier to handle errors if you call AllocateScreen manually.

Example
```
SES.AllocateScreen;
if SES.GetLastError <> 0 then begin
  WriteLn('Failed to allocate screen');
  Halt;
end;
```

After adding all entry fields and text fields, we call AllocateScreen to allocate an appropriate sized virtual screen.

See Also
CommandWindow.GetLastError Done

Declaration
```
destructor Done; virtual;
```

Purpose
Deallocate field list, screen buffers.

Description
This destructor disposes of the virtual screen associated with the ScrollingEntryScreen, then calls the parent's destructor (EntryScreen.Done) to dispose of the field list, screen buffers, etc.

Example
```
SES.Erase;
SES.Done;
```

Erase SES and release all heap space.

See Also
EntryScreen.Done

Init / InitCustom

Declaration

```
constructor Init(X1, Y1, X2, Y2 : Byte);
```

Purpose

Initialize the entry screen.

Description

This constructor instantiates a ScrollingEntryScreen using the default window options (DefWindowOptions) and the default color set (DefaultColorSet). See InitCustom for further details.

Example

```
if not SES.Init(25, 5, 55, 10) then begin
  WriteLn('Failed to init EntryScreen. Status = ', InitStatus);
  Halt;
end;
```

Instantiates SES using default window colors and options.

Declaration

```
constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                       Options : LongInt);
```

Purpose

Initialize the entry screen with custom window options.

Description

This constructor instantiates an EntryScreen using the specified window Options and Colors. X1, Y1, X2, and Y2 are the coordinates for the interior portion of the window.

InitCustom will fail only if the parent constructor (EntryScreen.InitCustom) fails. The most likely cause of failure is insufficient memory. The precise cause may be determined by examining InitStatus.

Several sets of attributes in the specified ColorSet have special relevance for window objects of type ScrollingEntryScreen. See the entries for AbstractSelector.InitCustom and EntryScreen.InitCustom for details.

Example

```
if not SES.InitCustom(25, 5, 55, 10,                {Window coordinates}
                      DefaultColorSet,              {ColorSet}
                      DefWindowOptions or wBordered) {Window options}
then begin
  WriteLn('Failed to init EntryScreen. Status = ', InitStatus);
  Halt;
end;
```

Instantiates SES in a bordered window with default colors.

See Also

AbstractSelector.InitCustom EntryScreen.InitCustom

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a scrolling entry screen from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to ScrollingEntryScreen.Store.

The various field types used by the ScrollingEntryScreen must be registered before calling Load, as shown in the example below. If appropriate, the address of the user record should be registered as well, and so should the addresses of any user-written routines associated with the ScrollingEntryScreen (pre-edit, post-edit, global validation, IncChoice).

It is not necessary to call AllocateScreen after a call to Load. Load does that for you.

The stream registration procedure for a ScrollingEntryScreen is ScrollingEntryScreenStream.

For further details, see "EntryScreens and Streams" in the introduction to this section.

Example

```
uses OpString, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEntry;
type
  InfoRec = record
              Name : String[30];
              Wage : Real;
              Misc : array[1..10] of String[10];
            end;
var
  Info : InfoRec;
  SES : ScrollingEntryScreen;
  S : BufIdStream;
begin
  FillChar(Info, SizeOf(InfoRec), 0);
  {Load SES from a stream}
  S.Init('ENTRYS.STM', SOpen, 1024);
  S.RegisterHier(ScrollingEntryScreenStream);
  S.RegisterHier(StringFieldStream);
  S.RegisterHier(RealFieldStream);
  S.RegisterPointer(1000, @Info);
  S.Get(SES);
  if S.GetStatus <> 0 then Halt;
  S.Done;
  ClrScr;
  SES.Process;
  SES.Done;
end.
```

Instantiates SES from a stream created by the Store example using the stream's Get method, which calls ScrollingEntryScreen.Load indirectly. Notice that, in addition to registering the ScrollingEntryScreen type, we must also register all of the field types used by it.

Unlike the Store example for the non-scrollable EntryScreen, the Store example for the ScrollingEntryScreen stored the address of the user record, so Load will not allocate heap space for it. So in order to reload the entry screen, we must register the address of a user record with RegisterPointer.

See Also

Store

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a scrolling entry screen in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the ScrollingEntryScreen to the stream S. The various field types used by the ScrollingEntryScreen must be registered before calling Store, as shown in the example below.

If appropriate, the address of the user record should be registered as well, and so should the addresses of any user-written routines associated with the ScrollingEntryScreen (pre-edit, post-edit, global validation, IncChoice).

To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

The stream registration procedure for a ScrollingEntryScreen is ScrollingEntryScreenStream.

For further details, see "EntryScreens and Streams" in the introduction to this section.

Example

```
uses OpString, OpCrt, OpRoot, OpCmd, OpFrame, OpWindow, OpSelect, OpEntry;
type
  InfoRec = record
              Name : String[30];
              Wage : Real;
              Misc : array[1..10] of String[10];
            end;
var Info : InfoRec;
    SES : ScrollingEntryScreen;
    S : BufIdStream;
    I, Status : Word;
begin
  if not SES.InitCustom(25, 5, 55, 10, DefaultColorSet,
                        DefWindowOptions or wBordered) then
    Halt;
  SES.AddStringField('Name:', 2, 1, '', 2, 7, 20, 0, Info.Name);
  SES.AddRealField('Wage:',3, 1, '99.99', 3, 7, 0, 0.0, 0.0, 2, Info.Wage);
  for I := 1 to 10 do
    SES.AddStringField('Misc'+Long2Str(I),I+3,1,'',I+3,7,10,0,Info.Misc[I]);
  SES.AllocateScreen;
  {Write the SES to a stream}
  SES.SetUserRecord(Info, SizeOf(Info));
  S.Init('ENTRYS.STM', SCreate, 1024);
  S.RegisterHier(ScrollingEntryScreenStream);
  S.RegisterHier(StringFieldStream);
  S.RegisterHier(RealFieldStream);
  S.RegisterPointer(1000, @Info);
  S.Put(SES);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status); Halt;
  end;
  S.Done;
  SES.Done;
end.
```

Instantiates SES, adds fields to it, and then stores SES in a stream using the stream's Put method, which calls Store indirectly. We call SetUserRecord to tell SES about the location and size of our data record, and we register its address to allow it to be located when the ScrollingEntryScreen is reloaded.

Note that, in addition to registering the ScrollingEntryScreen type, we've also had to register each of the field types used by it.

Note too that it is not necessary to call AllocateScreen in this particular case because SES is destroyed immediately after it is stored. See Load for an example program that reloads this ScrollingEntryScreen.

See Also

Load

VScreenToScreen

Declaration

```
procedure VScreenToScreen; virtual;
```

Purpose

Copy the virtual screen to the physical screen.

Description

This method is intended strictly for internal use, and should never be called directly. It is documented here only because it is one that you might want to override in an object derived from a ScrollingEntryScreen. What it does is copy the relevant portion of the virtual screen to the physical screen when the entry screen is being redrawn in its entirety or scrolled. But it is not the function of the method itself but rather the time at which it is called that makes it useful in derived types.

Suppose, for example, that you wanted an entry screen that scrolls horizontally to have a row of column labels displayed just above the first row of fields, and that these column labels needed to be displayed at all times, even if the entry screen has been scrolled vertically. (This rules out the use of text fields or a screen update procedure, since these hooks are used to display static text within the scrollable portion of the window.) Furthermore, the column labels need to change in unison with the contents of the entry screen as it is scrolled horizontally. (If you can't picture this situation, try popping up the quick reference chart in DESKPOP, then scrolling the window in all directions using the cursor keys. The row just below the border, which scrolls horizontally but not vertically, contains column labels.) The best way to accomplish this is to override the VScreenToScreen method, as shown in the sample code below.

Example

```
type
  MyEntryScreen = object(ScrollingEntryScreen)
    constructor InitCustom(X1, Y1, X2, Y2 : Byte; var Colors : ColorSet;
                           Options : LongInt);
    procedure VScreenToScreen; virtual;
  end;

  constructor MyEntryScreen.InitCustom(X1, Y1, X2, Y2 : Byte;
                                       var Colors : ColorSet; Options : LongInt);
  begin
    if not ScrollingEntryScreen.InitCustom(X1, Y1+2, X2, Y2, Colors,
                                           Options or wBordered) then
      Fail;
    {adjust frame}
    with wFrame do
      AdjustFrameCoords(frXL, frYL-2, frXH, frYH);
    if cwGetLastError = 0 then
      {add window divider}
      wFrame.AddSpanHeader(#195, #196, #180, 2, frTT);
    if cwGetLastError <> 0 then begin
      InitStatus := cwGetLastError;
      Done;
      Fail;
    end;
  end;
  procedure MyEntryScreen.VScreenToScreen;
  var S : string;
  begin
    ScrollingEntryScreen.VScreenToScreen;
    {...construct a full row of column labels in S...}
    fFastWrite(Copy(S, sesColOfs+1, Width), 1, 1, TextAttr);
  end;
```

MyEntryScreen is a specialized object type derived from ScrollingEntryScreen. The InitCustom constructor serves four purposes: 1) it insures that the VMT pointer for the object is set up correctly; 2) it insures that the window has a border; 3) it adjusts the coordinates for the inner portion of the window to set aside two rows above it that won't be scrolled; and 4) it adds a span header at the second row, leaving one row at the top for column labels.

VScreenToScreen does three things: 1) it calls the parent's VScreenToScreen method to update the scrollable portion of the window; 2) it constructs a full-sized row of column labels (i.e., one as wide as the virtual screen) and stores it in S; and 3) it displays the relevant portion of S on the screen row between the border and the span header.

Additional examples of this basic technique may be found in the OPCAL (Calendar object) and OPCALC (Calculator object) units. Although the objects in these units accomplish their tasks by overriding the UpdateContents method, the principle is the same.

See Also

SetScreenUpdateProc

G. OPEDIT: Advanced Line Editor

Like the EntryScreen, the LineEditor object supplied in OPEDIT works with the various field objects derived from EntryField. The similarities end there, however. A LineEditor is not descended from a CommandWindow, nor does it work with multiple data entry fields. Its purpose is simply to provide a high-level interface to the low-level field objects in OPFIELD, allowing them to be used to edit individual variables, as opposed to groups of variables.

As usual, we'll try to get you started with a simple example program:

```
{$V-}
program LineEditorExample; {EXLEDIT.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpEdit;
var
  LE : LineEditor;
  Name : String[40];
  Age : Integer;
begin
  ClrScr;
  Name := '';
  Age := 0;
  LE.Init(DefaultColorSet);
  LE.ReadString('Name: ', 4, 2, 40, 20, Name);
  if (LE.GetLastCommand <> ccQuit) and (LE.GetLastCommand <> ccError) then
    LE.ReadInteger('Age:  ', 6, 2, 3, 1, 150, Age);
  LE.Done;
end.
```

The first thing to notice here is the compiler directive {$V-}, without which the program cannot be compiled. All programs or units that use OPEDIT must be compiled with "relaxed" var string checking. ('{$V+}', the default setting, means "strict" var string checking.)

After clearing the screen and initializing the variables to be edited, the next step is to instantiate the line editor by calling Init. This is the only constructor for a LineEditor, and it takes a single parameter, a ColorSet. You'll notice that there is no test here to determine whether Init has failed, because Init does not need to allocate any memory on the heap. (All memory allocation/deallocation performed by a LineEditor is done within the edit methods.) Now we're ready to edit.

ReadString, one of several methods available for editing string variables, takes six parameters. The first is a prompt, which will be displayed immediately in front of the string being edited. The next two are the screen coordinates for the prompt (or the edit field itself, if there is no prompt). By default these coordinates are assumed to be absolute, but you can request that they be treated as window relative by enabling the sleWindowRelative option.

The fourth parameter is the maximum length of the string to be edited (Name). The fifth is the width of the "edit field," the region of the screen in which the string is displayed during editing. In most cases, these two parameters will be equal. But if the maximum length is greater than the width, the field editor will scroll the contents of the edit field horizontally as necessary. This feature allows strings of up to 254 characters to be edited, regardless of the width of the screen. In the case shown above, Name has a maximum length of 40, only 20 characters of which will be displayed at any one time.

After the first call to the ReadString method, GetLastCommand is called to determine what command the user issued to leave the editor. Unless mouse support is enabled, or user exit

commands have been added to the command table, the only exit commands it will return are ccQuit (<Esc>, <CtrlBreak>), ccSelect (<Enter>), or ccError. A return code of ccError generally indicates that there was insufficient memory for the edit method to create the EntryField to be edited. The precise cause of the error may be determined by calling GetLastError.

If the call to ReadString did not result in an exit command of ccQuit or ccError, the next step is to call ReadInteger to edit the Age variable. The first few parameters for ReadInteger are the same as those for ReadString: prompt, row, column. The next is the width of the edit field; there is no maximum length parameter for ReadInteger, which does not support horizontal scrolling. The final parameter is the Integer variable to edit, and the previous two specify the lower and upper bounds for the number, respectively. In this case, the user must enter a number between 1 and 150.

The final statement of the program, the call to the Done destructor, is technically unnecessary, for the same reason that Init cannot fail--there's nothing for Done to undo. Nevertheless, it's a good programming practice to always call an object's destructor when you're through with it. If you do so religiously, you're less likely to forget to make the call in cases where it really is necessary.

Our little test program is simple and straightforward, and it should take only a few moments of looking at it to figure what it does. There is, however, one important thing you can't know without running it, which is that the edit methods (ReadString and ReadInteger) do not save and restore the screen underneath the edit fields. That task isn't always necessary or desirable, so it is left to the programmer to perform when necessary.

Editing Methods

Because there are nearly 50 editing methods implemented by the LineEditor, they are discussed here in groups according to the field editor that they use, rather than individually. (For a discussion of field editors, see the documentation for OPFEDIT.) There is relatively little variation in the parameters passed to these methods, and once you've learned how to use one of them it is very easy to learn how to use the others, especially those in the same group.

The first three items in the parameter lists for all these methods are the same: Prompt, Row, and Col. The Prompt, if any, is a brief message to be displayed in front of the edit field, which follows it immediately. Row and Col give the coordinates for the Prompt. By default, these coordinates are assumed to be absolute, but if the sleWindowRelative option is enabled they will be construed as relative to the current window.

Most of the other options are identical (except for the two- to three-character prefix at the beginnings of their names) to their EntryField counterparts, which are discussed in the OPFIELD section. In a few cases, a particular option has special meaning for a certain method or methods, and these special cases are addressed below. But one other option has special meaning for all these methods, sleInsertByDefault. If this option is on, the LineEditor will reset its internal insert mode flag before calling the appropriate field editor. If it is false, the flag will retain its value between calls to editing methods.

For details about the behavior of a given field editor, see the OPFEDIT section of this chapter.

Read routines: SimpleStringEditor

The following methods use the simple string editor, which does not work with true picture masks. All of them are affected by three special options. If the leHouseCursorAtEnd option is on, the width of the edit field is effectively increased by one to make a place to "house" the

cursor when the field is full. The leForceUpper and leForceLower indicate that all characters entered should be converted to upper or lower case, respectively.

```
procedure ReadString(Prompt : string; Row, Col, MaxLen, Width : Byte;
                     var St : string);
  {-Edit a string}
procedure ReadSpecial(Prompt : string; Row, Col, MaxLen, Width : Byte;
                      PictureChar : Char; var St : string);
  {-Edit a string}
```

MaxLen gives the maximum length of the string to be edited (St), and Width specifies the width of the editing field. If MaxLen > Width, the field will be scrolled horizontally as necessary. ReadSpecial differs from ReadString only in that it takes a picture mask character that indicates what characters may be entered in the string. ReadString allows any character to be entered.

```
procedure ReadLongInt(Prompt : string; Row, Col, Width : Byte;
                      LLo, LHi : LongInt; var L : LongInt);
  {-Edit a LongInt}
procedure ReadInteger(Prompt : string; Row, Col, Width : Byte;
                      NLo, NHi : Integer; var N : Integer);
  {-Edit an Integer}
procedure ReadWord(Prompt : string; Row, Col, Width : Byte;
                   WLo, WHi : Word; var W : Word);
  {-Edit a Word}
procedure ReadByte(Prompt : string; Row, Col, Width : Byte;
                   BLo, BHi : Byte; var B : Byte);
  {-Edit a Byte}
procedure ReadShortInt(Prompt : string; Row, Col, Width : Byte;
                       SLo, SHi : ShortInt; var SI : Shortint);
  {-Edit a ShortInt}
```

Width is the width of the editing field. The xLo and xHi parameters specify the lower and upper limits for the number being edited (last parameter). If xHi = xLo, any number that is valid for the given data type may be entered.

```
procedure ReadReal(Prompt : string; Row, Col, Width, Places : Byte;
                   RLo, RHi : Real; var R : Real);
  {-Edit a Real}
{$IFDEF UseBcd}
procedure ReadBcd(Prompt : string; Row, Col, Width, Places : Byte;
                  BLo, BHi : Bcd; var B : BCD);
  {-Edit a BCD real}
{$ENDIF}
{$IFOPT N+}
procedure ReadExt(Prompt : string; Row, Col, Width, Places : Byte;
                  ELo, EHi : Extended; var E : Extended);
  {-Edit an Extended}
procedure ReadDbl(Prompt : string; Row, Col, Width, Places : Byte;
                  DLo, DHi : Double; var D : Double);
  {-Edit a Double}
procedure ReadSgl(Prompt : string; Row, Col, Width, Places : Byte;
                  SLo, SHi : Single; var S : Single);
  {-Edit a Single}
```

```
procedure ReadComp(Prompt : string; Row, Col, Width : Byte;
                   CLo, CHi : Comp; var C : Comp);
  {-Edit a Comp}
{$ENDIF}
```

Width is the width of the editing field. The xLo and xHi parameters specify the lower and upper limits for the number being edited (last parameter). If xHi = xLo, any number that is valid for the given data type may be entered. Places indicates the maximum number of decimal places to be displayed initially (e.g., if R = 1.2345 and Places = 2, then 1.23 would be displayed, and if the user accepted the field as is then R would equal 1.23 on exit). Note that values may be entered using scientific notation if desired, and that scientific notation will always be used to display the number initially if it is too large to be displayed using decimal notation.

Read routines: SimpleCharEditor

The following methods use the simple character editor rather than the simple string editor. Both are affected by several special options (besides leForceUpper and leForceLower). If leAllowEscape is in effect, the user is allowed to escape by giving the ccQuit command. If not, a valid response must be given. If leShowReadChar is in effect, the character entered by the user is displayed when it is entered; otherwise, it isn't. If leHideCursor is in effect, the cursor is hidden while waiting for the user to enter a response.

```
procedure ReadChar(Prompt : string; Row, Col : Byte;
                   ValidChars : CharSet; var Ch : Char);
  {-Display a prompt and wait for a key in ValidChars}
```

ValidChars specifies the characters that may be entered in the field. If the initial value of Ch is in ValidChars, and the leDefaultAccepted option is in effect, the prompt will have ' [' + Default + '] ' appended to it, and the user may accept the default response by pressing <Enter>. If not, the user must enter a valid character. If the leMapExtended option is in effect, the editor will allow extended keys (function keys, cursor keys, and all other keys for which the low byte of the key code is 0 and the high byte has the scan code) to be entered as responses. To signal an extended key, the editor will set the high bit of the character returned. For example, if the user presses <F1>, which has a key code of $3B00, Ch would be set to #$BB ($3B or $80). It would be accepted, of course, only if ValidChars contained #$BB.

```
function YesOrNo(Prompt : string; Row, Col : Byte; Default : Char) : Boolean;
  {-Return True for yes, False for no}
```

Default should usually be either 'Y' (YesChar) or 'N' (NoChar), in which case the prompt will have ' [' + Default + '] ' appended to it, and the user may accept the default response by pressing <Enter>. If Default is not 'Y' or 'N', the user must enter one or the other.

Read routines: StringEditor

Unlike the methods whose names start with 'Read', the following methods work with true picture masks. In a few cases, where there is a reasonable default, the Picture parameter is optional. These cases are addressed individually. If the comments following the declaration for a given method do not indicate that there is a default picture mask, there isn't one.

```
procedure EditString(Prompt : string; Row, Col : Byte; Picture : string;
                     Width : Byte; var St : string);
  {-Edit a String}
procedure EditArray(Prompt : string; Row, Col : Byte; Picture : string;
                    Width : Byte; var A);
  {-Edit an array of Char}
```

Width is the width of the edit field. If it is less than Length(Picture), the field will be scrolled horizontally as necessary during editing. If Picture is empty, a picture mask of Width 'X' characters will be assumed.

```
procedure EditLongInt(Prompt : string; Row, Col : Byte; Picture : string;
                      LLo, LHi : LongInt; var L : LongInt);
  {-Edit a LongInt}
procedure EditInteger(Prompt : string; Row, Col : Byte; Picture : string;
                      NLo, NHi : Integer; var N : Integer);
  {-Edit an Integer}
procedure EditWord(Prompt : string; Row, Col : Byte; Picture : string;
                   WLo, WHi : Word; var W : Word);
  {-Edit a Word}
procedure EditByte(Prompt : string; Row, Col : Byte; Picture : string;
                   BLo, BHi : Byte; var B : Byte);
  {-Edit a Byte}
procedure EditShortInt(Prompt : string; Row, Col : Byte; Picture : string;
                       SLo, SHi : ShortInt; var SI : Shortint);
  {-Edit a ShortInt}
```

The xLo and xHi parameters specify the lower and upper limits for the number being edited (last parameter). If xHi = xLo, any number that is valid for the given data type may be entered.

```
procedure EditReal(Prompt : string; Row, Col : Byte; Picture : string;
                   Places : Byte; RLo, RHi : Real; var R : Real);
  {-Edit a Real}
{$IFDEF UseBcd}
procedure EditBcd(Prompt : string; Row, Col : Byte; Picture : string;
                  Places : Byte; BLo, BHi : Bcd; var B : BCD);
  {-Edit a BCD real}
{$ENDIF}
{$IFOPT N+}
procedure EditExt(Prompt : string; Row, Col : Byte; Picture : string;
                  Places : Byte; ELo, EHi : Extended; var E : Extended);
  {-Edit an Extended}
procedure EditDbl(Prompt : string; Row, Col : Byte; Picture : string;
                  Places : Byte; DLo, DHi : Double; var D : Double);
  {-Edit a Double}
procedure EditSgl(Prompt : string; Row, Col : Byte; Picture : string;
                  Places : Byte; SLo, SHi : Single; var S : Single);
  {-Edit a Single}
procedure EditComp(Prompt : string; Row, Col : Byte; Picture : string;
                   CLo, CHi : Comp; var C : Comp);
  {-Edit a Comp}
{$ENDIF}
```

The xLo and xHi parameters specify the lower and upper limits for the number being edited (last parameter). If xHi = xLo, any number that is valid for the given data type may be entered. If Picture does not contain a decimal point, Places indicates the maximum number of decimal places to be displayed initially (e.g., if R = 1.2345 and Places = 2, then 1.23 would be displayed, and if the user accepted the field as is then R would equal 1.23 on exit). Note that values may not be entered using scientific notation with this editor, so the picture mask should not contain 'E'.

```
{$IFDEF UseDates}
procedure EditDate(Prompt : string; Row, Col : Byte; Picture : string;
                   DLo, DHi : Date; var D : Date);
  {-Edit a Date variable}
procedure EditDateSt(Prompt : string; Row, Col : Byte; Picture : DateString;
                     var DS : DateString);
  {-Edit a DateString}
procedure EditTime(Prompt : string; Row, Col : Byte; Picture : DateString;
                   TLo, THi : Time; var T : Time);
  {-Edit a Time variable}
{$ENDIF}
```

The xLo and xHi parameters specify the lower and upper limits for the value being edited (last parameter). If xHi = xLo, any number that is valid for the given data type may be entered. If Picture = '', then a picture mask of 'mm/dd/yy' is assumed for EditDate and EditDateSt, and 'hh:mm:ss' is assumed for EditTime.

Read routines: CharEditor

The methods shown below use CharEditor for editing. The picture mask passed to them should contain a single formatting character, one that is appropriate for the field type ('Y' for EditYesNo, for example). It may also contain any number of literal characters (e.g., '>Y<'). The last parameter in each case is the variable to be edited. In the case of EditChar, the ChLo and ChHi parameters specify the lower and upper limits for the character entered. If ChHi = ChLo, the picture mask alone determines what characters may be entered.

```
procedure EditChar(Prompt : string; Row, Col : Byte; Picture : string;
                   ChLo, ChHi : Char; var Ch : Char);
  {-Edit an array of Char}
procedure EditBoolean(Prompt : string; Row, Col : Byte; Picture : string;
                      var B : Boolean);
  {-Edit a Boolean variable}
procedure EditYesNo(Prompt : string; Row, Col : Byte; Picture : string;
                    var YN : Boolean);
  {-Edit a Yes-No variable}
```

Each of these methods has a default picture mask that is used if Picture = ''. For EditChar, it is 'X' (AnyChar). For EditBoolean, it is 'B' (BooleanOnly). And for EditYesNo it is 'Y' (YesNoOnly).

Read routines: NumberEditor

The following methods allow you to edit numeric variables using the NumberEditor. In each case, the last parameter is the variable to be edited, and the previous two parameters specify its lower and upper limits. If the upper limit is equal to the lower limit, any number that is valid for the given data type may be entered.

The Places parameter for the methods involving reals is currently ignored. If fractional numbers are to be allowed, the picture mask must contain a fixed decimal point. Future versions of **Object Professional** may make use of the Places parameter.

```
procedure EditNumericLongInt(Prompt : string; Row, Col : Byte;
                             Picture : string; LLo, LHi : LongInt;
                             var L : LongInt);
  {-Edit a LongInt (numeric)}
```

```
procedure EditNumericInteger(Prompt : string; Row, Col : Byte;
                             Picture : string; NLo, NHi : Integer;
                             var N : Integer);
  {-Edit an Integer (numeric)}
procedure EditNumericWord(Prompt : string; Row, Col : Byte; Picture : string;
                          WLo, WHi : Word; var W : Word);
  {-Edit a Word (numeric)}
procedure EditNumericByte(Prompt : string; Row, Col : Byte; Picture : string;
                          BLo, BHi : Byte; var B : Byte);
  {-Edit a Byte (numeric)}
procedure EditNumericShortInt(Prompt : string; Row, Col : Byte;
                              Picture : string; SLo, SHi : ShortInt;
                              var SI : Shortint);
  {-Edit a ShortInt (numeric)}
procedure EditNumericReal(Prompt : string; Row, Col : Byte; Picture : string;
                          Places : Byte; RLo, RHi : Real; var R : Real);
  {-Edit a Real (numeric)}
{$IFDEF UseBcd}
procedure EditNumericBcd(Prompt : string; Row, Col : Byte; Picture : string;
                         Places : Byte; BLo, BHi : Bcd; var B : BCD);
  {-Edit a BCD real (numeric)}
{$ENDIF}
{$IFOPT N+}
procedure EditNumericExt(Prompt : string; Row, Col : Byte; Picture : string;
                         Places : Byte; ELo, EHi : Extended; var E : Extended);
  {-Edit an Extended (numeric)}
procedure EditNumericDbl(Prompt : string; Row, Col : Byte; Picture : string;
                         Places : Byte; DLo, DHi : Double; var D : Double);
  {-Edit a Double (numeric)}
procedure EditNumericSgl(Prompt : string; Row, Col : Byte; Picture : string;
                         Places : Byte; SLo, SHi : Single; var S : Single);
  {-Edit a Single}
procedure EditNumericComp(Prompt : string; Row, Col : Byte; Picture : string;
                          CLo, CHi : Comp; var C : Comp);
  {-Edit a Comp (numeric)}
{$ENDIF}
```

Note that none of these has a default picture mask.

Others

The following edit methods let you take advantage of the special capabilities of the ChoiceField and MultiLineField objects, both of which have their own editors.

```
procedure EditChoice(Prompt : string; Row, Col : Byte; Picture : string;
                     DataSize : Byte; Increment : IncChoiceProc; var MC);
  {-Edit a multiple choice variable}
```

This method is the counterpart of EntryScreen.AddChoiceField. MC is a variable of any data type of up to 10 bytes that can be incremented and decremented (enumerated, integer, real, etc.). DataSize is the size of the variable. Increment has the form of an IncChoiceProc. See the discussion of "Multiple Choice Fields" in the OPENTRY chapter for details.

```
procedure EditMultiLine(Prompt : string; Row, Col : Byte;
                        Width, Height : Byte; var StArray);
  {-Edit a multi-line field}
```

This method is the counterpart of EntryScreen.AddMultiLineField, and allows you to edit an array of string. StArray is assumed to be an 'array[1..Height] of string[Width]'. See the discussion of "Multi-line Fields" in the OPENTRY chapter for details. Note that, like all the other edit methods, EditMultiLine *does not* save and restore the contents of the screen beneath the field.

Editing Commands

The editing commands supported by the LineEditor are arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. Because most of the commands behave differently depending on the field editor in use, the descriptions are generally preceded by one or more abbreviations, indicating the field editor being described. These are the abbreviations used, followed by the complete name of the field editor, and the corresponding LineEditor methods:

```
SS  SimpleStringEditor  ReadXxx
SC  SimpleCharEditor    ReadChar, YesOrNo
SE  StringEditor        EditXxx
CE  CharEditor          EditChar, EditBoolean, EditYesNo
NE  NumberEditor        EditNumericXxx
MC  ChoiceField.Edit    EditChoice
ML  MultiLine.Edit      EditMultiLine
```

The behavior of most of the cursor movement commands further depends on the state of the sleNoFieldMovement, which is on by default. If the option is in effect, no field movement command (ccLeft, ccRight, ccWordLeft, ccWordRight, ccUp, ccDown) can produce an exit. If the option is turned off, these commands will be treated just as they would if the field was part of an EntryScreen.

Cursor Movement

ccLeft **<Left>, <CtrlS>**
SS, SE, ML: Cursor left one character.

ccRight **<Right>, <CtrlD>**
SS, SE, ML: Cursor right one character.

ccWordLeft **<CtrlLeft>, <CtrlA>**
SS, SE, ML: Cursor left one word.

ccWordRight **<CtrlRight>, <CtrlF>**
SS, SE, ML: Cursor right one word.

ccHome **<Home>, <CtrlQ><S>**
SS, SE: Cursor to beginning of field.

ML: Cursor to beginning of line.

ccEnd **<End>, <CtrlQ><D>**
SS, SE: Cursor to end of field.

ML: Cursor to end of line.

ccUp **<Up>, <CtrlE>**
ML: If cursor is not on top line of edit field, moves cursor up one line.

ccDown **<Down>, <CtrlX>**
ML: If cursor not on bottom line of edit field, moves cursor down one line.

Insertion and Deletion

ccDel **<Del>, <CtrlG>**
SS, SE, NE, ML: Delete character at cursor.

ccBack **<Bksp>, <CtrlH>, <CtrlBksp>**
SS, SE, ML: Delete character to left of cursor.

NE: Delete character at cursor.

ccDelLine **<CtrlY>**
SS, SE, NE: Clear entire field.

ML: Delete current line.

ccDelEol **<CtrlEnd>, <CtrlQ><Y>**
SS, SE: Delete from cursor to end of subfield.

ML: Delete from cursor to end of line.

ccDelBol **<CtrlHome>**
SS, SE: Delete from beginning of subfield to cursor.

ML: Delete from beginning of line to cursor.

ccDelWord **<CtrlT>**
SS, SE: Delete word to right of cursor.

ML: Delete word to right of cursor. If cursor is beyond end of line, and following line will fit at end of current line, the following line is joined with the current line.

ccCtrlChar **<CtrlP>**
SS, SE, ML: Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is not available if the video attribute for control characters is the same as that for ordinary text and the leMapCtrls option is on.

ccIns **<Ins>, <CtrlV>**
SS, SE, ML: Toggle insert mode on and off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

Other

ccSelect **<Enter>, <CtrlM>, <ClickCenter>**
SS, SC, SE, CE, NE, MC: Accept contents of field.

ML: If in overtype mode, moves cursor to beginning of next line. If in insert mode, inserts new line at position of cursor and moves cursor to beginning of new line (if possible).

ccDone **<^Enter>, <CtrlJ>**
Accept contents of field.

ccMouseSel **<ClickLeft>**
SS, SC, SE, CE, NE, MC: Accept contents of field.

ML: If mouse within field, moves cursor to position of mouse.

ccRestore **<CtrlR>, <CtrlQ><L>**
SS, SC, SE, CE, NE, MC: Restore original contents of the field.

ML: Restore original contents of current line.

ccIncChoice **<+>, <Space>**
MC: Increment a multiple choice field. This command is meaningful only if EditChoice was called. In all other cases, <+> and <Space> will be treated as ordinary characters.

ccDecChoice **<->**
MC: Decrement a multiple choice field. This command is meaningful only if EditChoice was called. In all other cases, <-> will be treated as an ordinary character.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling EditCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a GetHelpProc, the line editor will pass it the following three parameters: ucEdit, nil, and the value designated as the line editor's help index (see LineEditor.SetHelpIndex), which defaults to 0.

ccQuit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit editing.

Primary Options

Most of the primary edit options recognized by the LineEditor correspond directly to the primary field options documented previously in the OPFIELD section. 'lePasswordMode', for example, means the same thing as 'efPasswordMode'. The meanings of such options are not repeated here. There are, however, several field options that are inappropriate for a LineEditor, and there are also a couple (of secondary options) that have meaning only for a LineEditor. These are the issues addressed below.

```
leProtected       = $00000001; {if set, field contents cannot be modified}
leHidden          = $00000002; {field is hidden}
leMapCtrls        = $00000004; {map control characters}
leMultChoice      = $00000008; {a multiple-choice field}
leInvisible       = $00000010; {if set, field is invisible}
```

With the exception of leMapCtrls, these field options are inappropriate for line editing. leMapCtrls is equivalent to the efMapCtrls option in OPFIELD.

```
leAutoAdvCharBegin = $00000020; {dummy--for completeness}
leAutoAdvCharEnd   = $00000040; {advance to next field--character entered}
leAutoAdvCurBegin  = $00000080; {advance to prev field--cursor key pressed}
leAutoAdvCurEnd    = $00000100; {advance to next field--cursor key pressed}
leAutoAdvanceChar  = leAutoAdvCharBegin+leAutoAdvCharEnd;
leAutoAdvanceCursor= leAutoAdvCurBegin+leAutoAdvCurEnd;
leAutoAdvanceBegin = leAutoAdvCharBegin+leAutoAdvCurBegin;
leAutoAdvanceEnd   = leAutoAdvCharEnd+leAutoAdvCurEnd;
leAutoAdvance      = leAutoAdvanceBegin+leAutoAdvanceEnd;
leInsertPushes     = $00000200; {inserting can push a character off end}
leRightJustify     = $00000400; {field is right-justified}
lePasswordMode     = $00000800; {contents of field are suppressed}
leCursorToEnd      = $00001000; {put cursor at end of string initially}
leTrimBlanks       = $00002000; {trim leading/trailing blanks when finished}
leClearFirstChar   = $00004000; {clear string if first character is ASCII}
leForceOvertype    = $00008000; {if ForceMode set, selects insert or overtype}
leForceMode        = $00010000; {force insert or overtype, else use default}
leClickExit        = $00020000; {produce ccClickExit on double click}
```

```
leBeepOnError      = $00040000; {beep when illegal char entered?}
leAutoNumLock      = $00080000; {auto-activate NumLock--numbers only}
leParensForMinus   = $00100000; {show numbers < 0 inside ()--numbers only}
```

These options all have the same meaning as their counterparts in OPFIELD (which have a prefix of 'ef' rather than 'le').

```
leExitAtEdges      = $00200000; {allow exit at edges--pick editor only}
leRequired         = $00400000; {if set, field cannot be empty}
```

These field options are inappropriate for line editing.

```
leHouseCursorAtEnd = $01000000; {extra char of width to hold cursor}
leForceUpper       = $02000000; {force chars to upper case?}
leForceLower       = $04000000; {force chars to lower case?}
```

These options apply only to the ReadXxx and YesOrNo methods. See the earlier section on read methods that use SimpleStringEditor.

```
leAllowEscape      = $08000000; {allow escape?}
leShowReadChar     = $10000000; {display the character pressed?}
leHideCursor       = $20000000; {hide cursor?}
leDefaultAccepted  = $40000000; {allow user to accept default?}
leMapExtended      = $80000000; {map extended keys?}
```

These options apply only to the ReadChar and YesOrNo methods. See the earlier section on read methods that use SimpleCharEditor.

Secondary Options

The secondary edit options shown below correspond to the secondary field options documented in the OPFIELD section ('sef' prefix). Unlike most of the primary options, however, most of the secondary options have particular significance for a LineEditor.

```
slePadCurrentOnly  = $08000000; {use pad character only when editing}
sleSuppressZero    = $10000000; {suppress display of 0's for integer types}
```

See the entry for the SetPadChar method for a discussion of slePadCurrentOnly. The sleSuppressZero, which has the same meaning as its counterpart in OPFIELD, is generally inappropriate for line editing.

```
sleNoFieldMovement = $20000000; {don't allow field movement commands to exit}
```

See the introduction to the section on Editing Commands, above, for a discussion of this option.

```
sleWindowRelative  = $40000000; {coords treated relative to current window?}
sleInsertByDefault = $80000000; {default to insert mode?}
```

See the introduction to the section on Editing Methods, above, for a discussion of these two options.

Constants

```
BadEditOptions     : LongInt = leProtected+leHidden+leMultChoice+leInvisible+
                               leClickExit+leExitAtEdges+leRequired;
```

Primary options that exist for internal use, or are inappropriate while using a LineEditor, and may not be altered by calling leEditOptionsOn or leEditOptionsOff.

```
BadSecEditOptions  : LongInt = ifBoolean+ifNumeric+ifSimple+ifNested+
                               ifMultiLine+ifModified+ifNotNext+ifNotPrev+
                               ifNoMouseExit+ifSuppressFirst+ifNoLiterals+
                               ifEditing+ifSemiHidden+ifRealVar;
```

Secondary options that exist for internal use, or are inappropriate while using a LineEditor, and may not be altered by calling leSecEditOptionsOn or leSecEditOptionsOff.

```
DefEditOptions     : LongInt = leCursorToEnd+leTrimBlanks+leClearFirstChar+
                               leMapCtrls+leHouseCursorAtEnd;
```

The default primary options for a LineEditor.

```
DefSecEditOptions  : LongInt = sleNoFieldMovement+sleInsertByDefault+
                               ifNoMouseExit;;
```

The default secondary options for a LineEditor.

```
EditKeyMax = 180;
EditKeyID  : string[15] = 'opedit keys 1.0';
EditKeySet : array[0..EditKeyMax] of Byte = (...);
EditCfgEnd : Byte = 0;
```

EditKeyId is an ID string used to mark the the beginning of the configuration data area for OPEDIT; EditCfgEnd marks the end of the configuration data area. In between them is EditKeySet, the command table used by EditCommands. EditKeyMax is the last valid index into the table.

Types

```
LineEditor =
  object(Root)
    ...
    leInsertMode  : Boolean;    {True if in insert mode; False for overtype}
    leColors      : ColorSet;   {colors}
    ...
  end;
```

A line editor. leInsertMode indicates whether or not the LineEditor is currently in insert mode. leColors is a copy of the ColorSet passed to the Init constructor. (See the entry for Init.)

Variables

```
EditCommands : CommandProcessor;
```

The default command processor for a LineEditor.

GetLastCommand / GetLastError / GetLastKey

Declaration
```
function GetLastCommand : Word;
```

Purpose
Return last command entered by user.

Description
This method returns the last command entered by the user. It is typically called immediately after a call to an edit method. If the command returned is ccError, there was a fatal error within the edit method; the error code may be obtained by calling GetLastError.

Example
```
LE.ReadString('Name: ', 4, 2, 40, 20, Name);
if LE.GetLastCommand = ccQuit then
  Exit;
```
Exits if the last command entered by the user was ccQuit.

See Also
GetLastKey

Declaration
```
function GetLastError : Word;
```

Purpose
Get last error code.

Description
This method should be called only if GetLastCommand returns an exit code of ccError, indicating that a fatal error occurred within an edit method. Currently the only possible error code in such a case is epFatal+ecOutOfMemory, which indicates that the edit method was unable to instantiate the EntryField that it needed to create in order to do any editing.

Example
```
LE.ReadString('Name: ', 4, 2, 40, 20, Name);
case LE.GetLastCommand of
  ccQuit  : Exit;
  ccError : WriteLn('Error during read: ', GetLastError);
end;
```
Reports the error code if the call to ReadString fails.

See Also
GetLastCommand SetErrorProc

Declaration
```
function GetLastKey : Word;
```

Purpose
Return last keystroke entered by user.

Description
This method returns the last keystroke entered by the user. This information is generally less useful than that provided by GetLastCommand.

See Also
GetLastCommand

Declaration
```
constructor Init(var Colors : ColorSet);
```

Purpose
Sets default values for attributes, etc.

Description
This constructor initializes the object in preparation for a call to an edit method. Note that this constructor will never fail, since it does not have to allocate any memory from the heap.

Although the LineEditor stores a complete copy of the ColorSet passed to it as the Colors parameter, only three sets of attributes in the specified ColorSet actually have any special relevance for it. SelFieldColor/SelFieldMono determine the attribute used to display ordinary text with a string being edited. CtrlColor/CtrlMono determine the attribute used to display control characters embedded in the text. SelPromptColor/SelPromptMono determine the colors used to display prompts.

After the object has been instantiated, these attributes may be changed by modifying the appropriate fields in the ColorSet (leColors) stored within the LineEditor:

```
LE.leColors.SetSelectedFieldAttr(...);
LE.leColors.SetCtrlAttr(...);
LE.leColors.SetSelectedPromptAttr(...);
```

Example
```
LE.Init(DefaultColorSet);
```
Instantiate a line editor using the default color set (can't fail).

See Also
ColorSet.SetXxxAttr

Declaration
```
function leEditOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if all of the specified primary options are currently selected.

Example
```
if SLE.leEditOptionsAreOn(leBeepOnError) then
  SLE.leEditOptionsOff(leBeepOnError)
else
  SLE.leEditOptionsOn(leBeepOnError);
```
Toggle the state of the leBeepOnError option.

See Also
leEditOptionsOff leEditOptionsOn

leEditOptionsOff / leEditOptionsOn

Declaration
```
procedure leEditOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified primary option(s), excluding those designated as BadEditOptions (see the Constants section, above).

Example
```
LE.leEditOptionsOff(leTrimBlanks);
```
Turn off options to trim blanks.

See Also
leEditOptionsAreOn

Declaration
```
procedure leEditOptionsOn(OptionFlags : LongInt);
```

Purpose
Activate multiple options.

Description
This method activates the specified primary option(s), excluding those designated as BadEditOptions (see the Constants section, above).

Example
```
LE.leEditOptionsOn(leTrimBlanks);
```
Turns on option to trim blanks.

See Also
leEditOptionsAreOn

Declaration
```
function leSecEditOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if all of the specified secondary options are currently selected.

Example
```
if LE.leSecEditOptionsAreOn(sleInsertByDefault) then
  LE.leSecEditOptionsOff(sleInsertByDefault)
else
  LE.leSecEditOptionsOn(sleInsertByDefault);
```
Toggle the state of the sleInsertByDefault option.

See Also
leSecEditOptionsOn

Declaration
```
procedure leSecEditOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified secondary option(s), excluding those designated as BadSecEditOptions (see the Constants section, above).

Example
```
LE.leSecEditOptionsOff(sleWindowRelative);
```
Turns off the option to treat screen coordinates as window-relative.

See Also
leSecEditOptionsOn

leSecEditOptionsOn / Load

Declaration

```
procedure leSecEditOptionsOn(OptionFlags : LongInt);
```

Purpose

Activate multiple options.

Description

This method activates the specified secondary option(s), excluding those designated as BadSecEditOptions (see the Constants section, above).

Example

```
LE.leSecEditOptionsOn(sleNoFieldMovement);
```

Turn on option to suppress field movement commands.

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a line editor from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to LineEditor.Store.

The stream registration procedure for a LineEditor is LineEditorStream.

Example

```
uses OpCrt, OpRoot, OpCmd, OpEdit;
var
  LE   : LineEditor;
  S    : BufIdStream;
  Name : String[40];
  Age  : Integer;
begin
  {Initialize LineEditor from stream}
  S.Init('LINEEDIT.STM', SOpen, 1024);
  S.RegisterHier(LineEditorStream);
  S.Get(LE);
  if S.GetStatus <> 0 then Halt;
  S.Done;
  {Continue working with LE}
  Name := '';
  Age := 0;
  LE.ReadString('Name: ', 4, 2, 40, 20, Name);
  if LE.GetLastCommand <> ccQuit then
    LE.ReadInteger('Age:  ',6, 2, 3, 1, 150, Age);
  LE.Done;
end.
```

In this example, we load LE from a stream created by the Store example using the stream's Get method, which calls LineEditor.Load indirectly.

See Also

Store

Declaration

```
procedure SetCommandProcessor(var CP : CommandProcessor);
```

Purpose

Set command processor to use.

Description

This method allows you to specify an alternate command processor to be used in conjunction with the line editor.

Example

```
var
  LE : LineEditor;
  MyCommandProcessor : CommandProcessor;
begin
  MyCommandProcessor.Init(...);
  LE.Init(DefaultColorSet);
  LE.SetCommandProcessor(MyCommandProcessor);
end.
```

Sets the command processor to MyCommandProcessor.

See Also

CommandProcessor.Init

SetErrorProc

Declaration

```
procedure SetErrorProc(EEP : ErrorProc);
```

Purpose

Set procedure to call on error.

Description

This method lets you specify a procedure to be called when an error occurs during editing. Typically, any errors reported to the error handler are simply warnings generated by the routines that validate user input ('Value not in valid range', 'Invalid number', etc.).

The error handler must be of the following form:

```
{$F+}
procedure MyErrHan(UnitCode : Byte; var ErrorCode : Word; ErrorMsg : string);
begin
end;
```

UnitCode is a unit code (ucEdit) that indicates the source of the error. ErrorCode identifies the type of error that occurred. ErrorMsg is a string that describes the error. For additional information on error handlers, see the entry for CommandWindow.SetErrorProc (OPWINDOW). See Appendix D for a listing of standard error codes.

Example

```
uses
  OpCrt, OpString, OpRoot, OpCmd, OpEdit;
var
  LE : LineEditor;
  Age : Integer;

  {$F+}
  procedure MyErrHan(UnitCode : Byte; var ErrorCode : Word; ErrorMsg : string);
  begin
    RingBell;
    GotoXY(1, 1);
    if ErrorCode div 10000 = etWarning then
      Write(ErrorMsg)
    else
      Write('Error: ', ErrorCode, ' ', ErrorMsg);
  end;
  {$F-}

begin
  {Initialize LineEditor and fields}
  LE.Init(DefaultColorSet);
  Age := 0;
  LE.SetErrorProc(MyErrHan);
  LE.ReadInteger('Age:  ',6, 2, 3, 1, 150, Age);
  LE.Done;
end.
```

Installs a simple error procedure called MyErrorProc. This error procedure rings a bell and displays the passed message. Note that it discriminates between real errors and user-input errors (warnings).

See Also

GetLastError

Declaration
```
procedure SetHelpIndex(Index : Word);
```

Purpose
Set the help index.

Description
This method allows you to set the help index for the line editor. This value is passed to the GetHelpProc associated with the line editor's CommandProcessor when the ccHelp command is issued.

Example
```
LE.SetHelpIndex(101);
```
Sets the help index for the line editor to 101.

Declaration
```
procedure SetPadChar(Ch : Char);
```

Purpose
Set character used to pad strings being edited.

Description
This method allows you to specify the character used to pad the end of a field being edited. For example, if you had a field 20 characters wide and a pad character of '_', the field would look something like this on the screen:

```
Name: John Smith_________
```

Note that the pad character affects only the way that the field is displayed. It does not affect the actual contents of the field.

Example
```
LE.SetPadChar('_');
LE.leSecEditOptionsOn(slePadCurrentOnly);
```
Uses underscore (instead of ' ') as the pad character. The slePadCurrentOnly indicates that the underscores should appear only while editing. After the user has accepted the contents of the field, it would be redrawn with a pad character of ' '.

Declaration
```
procedure SetPasswordChar(Ch : Char);
```

Purpose
Set character used to represent text in password mode.

Description
This method allows you to specify the character used to represent ordinary text within a field while in password mode. For example, suppose we have a picture mask of '999-999-9999' (for a phone number field), a pad character of ' ', and a password character of '*'. Here's what the field would look like in both normal and password modes at different stages of the entry process:

```
Normal:    ___-___-____   415-55_-____   415-555-12__   415-555-1212
Password:  ___-___-____   ***-**_-____   ***-***-**__   ***-***-****
```

Example
```
LE.SetPasswordChar('*');
LE.leEditOptionsOn(lePasswordMode);
LE.ReadString('Name: ', 4, 2, 40, 40, Name);
```
Edit Name in password mode, with each character entered being represented as an asterisk.

Store

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a line editor in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream). Store writes an image of the LineEditor to the stream S. To check for errors that may have occurred during the Store, call the stream's GetStatus function as shown in the example below.

Note that, if you have installed a special error handler by calling SetErrorProc or have specified an alternate command processor by calling SetCommandProcessor, you should register their addresses by calling the stream's RegisterPointer method before calling Put (or Store).

The stream registration procedure for a LineEditor is LineEditorStream.

Example

```
uses
  OpCrt, OpRoot, OpCmd, OpEdit;
var
  LE : LineEditor;
  S : BufIdStream;
  Status : Word;
begin
  LE.Init(DefaultColorSet);
  S.Init('LINEEDIT.STM', SCreate, 1024);
  S.RegisterHier(LineEditorStream);
  S.Put(LE);
  Status := S.GetStatus;
  if Status <> 0 then begin
    WriteLn('Store error ', Status);
    Halt;
  end;
  S.Done;
  LE.Done;
end.
```

Instantiates LE, then stores it using the stream's Put method, which calls Store indirectly. See Load for an example program that reloads this LineEditor.

See Also

Load

H. OPSEDIT: Simple Line Editor

Despite all its virtues, or perhaps because of them, OPEDIT has one potentially important drawback: it causes quite a bit of general purpose code to be linked into a program. In larger applications, especially those that use OPENTRY, the amount of code that it adds to a program is generally acceptable. But for some applications, OPEDIT's power and flexibility come at too high a price. If you're willing to sacrifice a host of features in exchange for a substantial decrease in code size, OPSEDIT is your best alternative.

OPSEDIT provides a simplified version of the LineEditor called, aptly enough, the SimpleLineEditor. It offers far fewer options than the LineEditor, and it works only with string variables, but it does implement the same set of basic editing commands and most of the more commonly used options.

To get you started with OPSEDIT, let's examine the following program:

```
{$V-}
program SimpleEditorExample;  {EXSEDIT.PAS}
uses
  OpCrt, OpRoot, OpCmd, OpSEdit;
var
  SLE  : SimpleLineEditor;
  Name : string[40];
  Age  : string[3];
begin
  ClrScr;
  Name := '';
  Age := '';
  SLE.Init(DefaultColorSet);
  SLE.ReadString('Name: ', 4, 2, 40, 20, Name);
  if SLE.GetLastCommand <> ccQuit then
    SLE.ReadString('Age:  ', 6, 2, 3, 3, Age);
  SLE.Done;
end.
```

The first thing you need to notice here is the {$V-} directive at the top of the program. As with OPEDIT, any program or unit that uses OPSEDIT must be compiled with "relaxed" var string checking.

After clearing the screen and initializing the string variables to be edited, the next step is to instantiate the line editor by calling Init. This is the only constructor for a SimpleLineEditor, and like LineEditor.Init it takes a single parameter, a ColorSet. You'll notice that there is no test here to determine whether Init has failed, because it can't fail. Init does not need to allocate any memory on the heap, so there's no reason for it to fail.

Now we're ready to edit. ReadString takes six parameters. The first is a prompt, which will be displayed immediately in front of the string being edited. The next two parameters are the screen coordinates for the prompt (or the edit field, if there is no prompt). By default these coordinates are assumed to be absolute, but you can request that they be treated as window relative by enabling the seWindowRelative option.

The fourth parameter is the maximum length of the string to be edited (the final parameter, Name in this case). The fifth is the width of the edit field. Normally the two will be equal. As in OPEDIT, however, if the maximum length is greater than the width, the contents of field will be scrolled horizontally as necessary. That will be the case here, where Name has a maximum length of 40, only 20 characters of which will be displayed at any one time.

After the first call to the ReadString method, GetLastCommand is called to determine what command the user issued to leave the line editor. Unless mouse support is enabled, or user exit commands have been added to the command table, the only exit commands it will return are ccQuit (<Esc> or <CtrlBreak>) or ccSelect (<Enter>). In this case, the second call to ReadString is skipped if the first call resulted in an exit command of ccQuit.

The call to the Done destructor at the end of the program is actually unnecessary, but is included on general principles.

If you're curious to see how much code space is saved by using OPSEDIT rather than OPEDIT, you only need to make two changes to the sample program above: 1) change "OpSEdit" to "OpEdit" in the Uses list, and 2) change SLE from a "SimpleLineEditor" to a "LineEditor. You should see an increase in program size of roughly 9K by switching to OPEDIT.

Editing Commands

The editing commands supported by the SimpleLineEditor are arranged by category in the list below. In each case the first line gives the name of the command, followed by the key(s) to which it is normally assigned. The second and following lines give a brief description of the command. The default key assignments are stored in the global typed constant SimpEditKeySet.

Cursor Movement

ccLeft **<Left>, <CtrlS>**
Cursor left one character.

ccRight **<Right>, <CtrlD>**
Cursor right one character.

ccWordLeft **<CtrlLeft>, <CtrlA>**
Cursor left one word.

ccWordRight **<CtrlRight>, <CtrlF>**
Cursor right one word.

ccHome **<Home>, <CtrlQ><S>**
Cursor to beginning of line.

ccEnd **<End>, <CtrlQ><D>**
Cursor to end of line.

Insertion and Deletion

ccDel **<Del>, <CtrlG>**
Delete character at cursor.

ccBack **<Bksp>, <CtrlH>, <CtrlBksp>**
Delete character to left of cursor.

ccDelLine **<CtrlY>**
Delete entire line.

ccDelEol **<CtrlEnd>, <CtrlQ><Y>**
Delete from cursor to end of line.

ccDelBol **<CtrlHome>**
Delete from start of line to cursor.

ccDelWord **<CtrlT>**
Delete word to right of cursor.

ccCtrlChar **<CtrlP>**
Insert control character. For example, to insert a ^G, you would enter <CtrlP><CtrlG>. This command is not available if the video attribute for control characters is the same as that for ordinary text and the seMapCtrls option is on.

ccIns **<Ins>, <CtrlV>**
Toggle insert mode on and off. A fat cursor indicates insert mode; a thin cursor indicates overtype mode.

Other

ccSelect **<Enter>, <CtrlM>, <ClickCenter>**
Accept contents of line.

ccMouseSel **<ClickLeft>**
Same as ccSelect.

ccRestore **<CtrlR>, <CtrlQ><L>**
Restore original contents of the current line.

ccHelp **<F1>, <ClickBoth>**
Help. If a user-written help routine has been established by calling SimpEditCommands.SetHelpProc, pressing <F1> will call that routine; otherwise this command does nothing. If there is a GetHelpProc, the line editor will pass it the following three parameters: ucSEdit, nil, and the value designated as the line editor's help index (see SimpleLineEditor.SetHelpIndex), which defaults to 0.

ccQuit **<Esc>, <CtrlBreak>, <ClickRight>**
Quit editing.

Constants

```
BadSimpEditOptions : LongInt = 0;
```

Options that exist for internal use, and may not be altered by calling seOptionsOn or seOptionsOff. No options currently fall into this category.

```
DefSimpEditOptions : LongInt = seCursorToEnd+seTrimBlanks+seClearFirstChar+
                               seMapCtrls+seHouseCursorAtEnd+
                               seInsertByDefault;
```

The default options for a SimpleLineEditor.

```
seMapCtrls         = $00000001; {map control characters}
seCursorToEnd      = $00000002; {put cursor at end of string initially}
seTrimBlanks       = $00000004; {trim leading/trailing blanks when finished}
seClearFirstChar   = $00000008; {clear string if first character is ASCII}
seBeepOnError      = $00000010; {beep when illegal char entered?}
seHouseCursorAtEnd = $00000020; {extra space to hold cursor when string full?}
seForceUpper       = $00000040; {force chars to upper case?}
seForceLower       = $00000080; {force chars to lower case?}
seWindowRelative   = $00000100; {coords treated relative to current window?}
seInsertByDefault  = $00000200; {default to insert mode?}
```

Options available for a SimpleLineEditor. seMapCtrls indicates that any control characters in the string being edited should be mapped to uppercase alphabetics (^A to 'A', for

example) and displayed in a different video attribute. seCursorToEnd indicates that, by default, the cursor should initially be positioned just beyond the last character in the line. seTrimBlanks indicates that any blank spaces at the end of the line should be stripped. seClearFirstChar indicates that the existing contents of the line should be cleared if the first key pressed during editing is a character.

seBeepOnError indicates whether or not the editor should beep (call the RingBell routine in OPCMD) when the user enters an invalid command or tries to enter a character when the line is already full. seHouseCursorAtEnd indicates that an extra space should be reserved just beyond the end of the edit field to house the cursor when the line is full. seForceUpper indicates that all characters entered by the user should be forced to upper case. seForceLower indicates that all characters entered by the user should be forced to lower case. seWindowRelative indicates that all screen coordinates passed to the line editor should be considered relative to the current window. seInsertByDefault indicates that insert mode should be activated each time that the ReadString method is called.

```
SimpEditKeyMax = 150;
SimpEditKeyID : string[16] = 'opsedit keys 1.0';
SimpEditKeySet : array[0..SimpEditKeyMax] of Byte = (...);
SimpEditCfgEnd : Byte = 0;
```

SimpEditKeyId is an ID string used to mark the the beginning of the configuration data area for OPSEDIT; SimpEditCfgEnd marks the end of the configuration data area. In between them is SimpEditKeySet, the command table used by SimpEditCommands. SimpEditKeyMax is the last valid index into the table.

Types

```
SimpleLineEditor =
  object(Root)
    ...
  end;
```

A simple line editor.

Variables

```
SimpEditCommands : CommandProcessor;
```

Default command processor for a SimpleLineEditor.

Declaration
```
function GetLastCommand : Word;
```

Purpose
Return last command entered by user.

Description
This method returns the last command entered by the user. It is typically called immediately after a call to ReadString.

Example
```
SLE.ReadString('Name: ', 4, 2, 40, 20, Name);
if SLE.GetLastCommand = ccQuit then
  Exit;
```
Exits if the last command entered by the user was ccQuit.

See Also
ReadString

Declaration
```
function GetLastKey : Word;
```

Purpose
Return last keystroke entered by user.

Description
This method returns the last keystroke entered by the user. This information is generally less useful than that provided by GetLastCommand.

Init / ReadString

Declaration

```
constructor Init(var Colors : ColorSet);
```

Purpose

Sets default values for attributes, etc.

Description

This constructor initializes the object in preparation for a call to ReadString. Note that this constructor will never fail, since it does not have to allocate any memory from the heap.

Three sets of attributes in the specified ColorSet have special relevance for objects of this type. SelFieldColor/SelFieldMono determine the attribute used to display ordinary text with a string being edited. CtrlColor/CtrlMono determine the attribute used to display control characters embedded in the text. SelPromptColor/SelPromptMono determine the colors used to display prompts. After the object has been instantiated, these attributes may be changed by calling the SetFieldAttr, SetCtrlAttr, and SetPromptAttr methods, respectively.

Example

```
SLE.Init(DefaultColorSet);
```

Instantiate a simple line editor using the default color set (can't fail).

See Also

SetCtrlAttr
SetFieldAttr
SetPromptAttr

Declaration

```
procedure ReadString(Prompt : string; Row, Col : Byte;
                     MaxLen, Width : Byte; var S : string);
```

Purpose

Edit a string.

Description

This is the method that actually performs all the line editing functions.

Prompt is a string to be displayed on the screen at Row,Col. (Row and Col are treated as absolute unless the seWindowRelative option is in effect.) S, the string to be edited, is displayed immediately to its right. MaxLen is the maximum length of the string; it must be in the range 1..254. Width is the width of the edit field (the region of the screen in which editing takes place).

In most cases, MaxLen should equal Width. If MaxLen is greater than Width, then the edit field will be scrolled horizontally as necessary when editing portions of the string beyond S[Width]. MaxLen should never be less than Width. If it is, Width will be reset to MaxLen.

Example

```
SLE.ReadString('Name: ', 4, 2, 40, 20, Name);
if SLE.GetLastCommand = ccQuit then
  Exit;
```

Reads a 40-character string within a 20-character edit field.

See Also

GetLastCommand

Declaration
```
function seOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose
Return True if all specified options are on.

Description
This method returns True if all of the specified options are currently selected.

Example
```
if SLE.seOptionsAreOn(seBeepOnError) then
  SLE.seOptionsOff(seBeepOnError)
else
  SLE.seOptionsOn(seBeepOnError);
```
Toggle the state of the seBeepOnError option.

Declaration
```
procedure seOptionsOff(OptionFlags : LongInt);
```

Purpose
Deactivate multiple options.

Description
This method deactivates the specified option(s), excluding those designated as BadSimpEditOptions (see the Constants section, above).

Example
```
SLE.seOptionsOff(seTrimBlanks+seForceUpper);
```
Turns off the options to trim blanks and force upper case.

Declaration
```
procedure seOptionsOn(OptionFlags : LongInt);
```

Purpose
Activate multiple options.

Description
This method activates the specified option(s), excluding those designated as BadSimpEditOptions (see the Constants section, above).

Example
```
SLE.seOptionsOn(seTrimBlanks);
```
Turns on the option to trim blanks.

SetCommandProcessor / SetCtrlAttr / SetFieldAttr

Declaration

```
procedure SetCommandProcessor(var CP : CommandProcessor);
```

Purpose

Set command processor to use.

Description

This method allows you to specify an alternate command processor to be used in conjunction with the line editor.

Example

```
var
  SLE : SimpleLineEditor;
  MyCommandProcessor : CommandProcessor;
begin
  MyCommandProcessor.Init(...);
  SLE.Init(DefaultColorSet);
  SLE.SetCommandProcessor(MyCommandProcessor);
end.
```

Sets the command processor to MyCommandProcessor.

See Also

CommandProcessor.Init

Declaration

```
procedure SetCtrlAttr(Color, Mono : Byte);
```

Purpose

Set attributes for control characters.

Description

This method allows you to set the attributes used to display control characters. If the seMapCtrls option is off, these attributes should normally be the same as those for ordinary text. If it is on, as it is by default, the attributes must be different, so that the user can distinguish ^A from 'A', etc.

Example

```
SLE.SetCtrlAttr($03, $70);
```

Sets the control-character attribute to cyan on black for color monitors, reverse video for monochrome monitors.

Declaration

```
procedure SetFieldAttr(Color, Mono : Byte);
```

Purpose

Set attributes for line being edited.

Description

This method allows you to set the attributes used to display normal text within the line being edited.

Example

```
SLE.SetFieldAttr($03, $07);
```

Sets the field attribute to cyan on black for color monitors, normal video for monochrome monitors.

See Also

SetPromptAttr

Declaration

```
procedure SetHelpIndex(Index : Word);
```

Purpose

Set the help index.

Description

This method allows you to set the help index for the line editor. This value is passed to the GetHelpProc associated with the line editor's CommandProcessor when the ccHelp command is issued.

Example

```
SLE.SetHelpIndex(101);
```

Sets the help index for the line editor to 101.

Declaration

```
procedure SetPromptAttr(Color, Mono : Byte);
```

Purpose

Set attributes for prompts.

Description

This method allows you to set the attributes used to display the prompt associated within a line being edited.

Example

```
SLE.SetPromptAttr($0E, $0F);
```

Sets the prompt attribute to yellow on black for color monitors, intense video for monochrome monitors.

See Also

SetCtrlAttr SetFieldAttr

I. OPDATE: Date and Time Variables

OPDATE specializes in history. Its primary function is to provide routines that allow you to store dates and times in compact formats, to convert them to other forms when necessary, and to perform date/time arithmetic.

One of its chief strengths is that OPDATE provides numerous functions for converting Julian dates and times to strings and back again. To allow the maximum amount of flexibility, and to insure that the same routines may be used internationally, OPDATE lets you specify a "picture mask" to indicate how a given string should be formatted. For example, let's say that a Julian date variable contains January 2, 1900 and you want to convert it to a string (using the DateToDateString routine). There are several possible picture masks you could use:

```
Picture                 Result
-----------------------------------
'mm/dd/yy'              '01/02/00'
'MM/dd/yyyy'            ' 1/02/1900'
'dd/mm/yyyy'            '02/01/1900'
'DD/mm/yyyy'            ' 2/01/1900'
'dd-nnn-yyyy'           '02-Jan-1900'
'dd-NNN-yyyy'           '02-JAN-1900'
'www dd nnn yyyy'       'Tue 02 Jan 1900'
'WWW NNN dd yyyy'       'TUE JAN 02 1900'
```

Time strings are handled in similar fashion, except that special formatting characters are available for working with times in am/pm (12-hour) format as well as military (24-hour) format. For example, let's say that you wanted to convert a Time variable containing 1:02 pm (13:02) into a string using TimeToTimeString. Here are some possible picture masks:

```
Picture                 Result
-----------------------------------
'hh:mm:ss'              '13:02:00'
'hh:mm'                 '13:02'
'hh:mm te'              '01:02 pm'
'hh:mmt'                '01:02p'
```

Note that string representations of time are always assumed to be in military format unless a 't' appears in the picture mask. An 'e' (em) in the mask should always be accompanied by a corresponding 't' (time).

Now that you have a general idea of what picture masks do, let's look at the characters that can appear in them. The following special characters are meaningful in picture masks:

```
Char        Marks location of or is replaced by
-------------------------------------------------------------------
'm', 'M'    month (date fields only)
'd', 'D'    day (date)
'y'         year (date)
'n', 'N'    month name (date)
'w', 'W'    day of week name (date)
'/'         '/' or SlashChar (date; semi-literal)

'h', 'H'    hours (time fields only)
'm', 'M'    minutes (time)
's', 'S'    seconds (time)
':'         ':' or ColonChar (time; semi-literal)
't'         'p', 'P', 'a', 'A' (time; case depends on UpCaseTime)
'e'         'm' or 'M' (time; case depends on UpCaseTime)
```

In most cases, the difference between the uppercase and lowercase formatting characters is that the lowercase ones cause numbers to be padded with '0', while the uppercase ones cause them to be padded with spaces. For example, if the date is 1/1/1900, a picture mask of 'MM/dd/yyyy' would produce ' 1/01/1900', while a mask of 'mm/dd/yyyy' would produce '01/01/1900'. 'Y' is not allowed as a formatting character because 1) it would probably never be used, and 2) it would create a conflict with one of the formatting characters in OPFIELD, which depends on OPDATE.

The characters 'n' and 'N' may be used in place of 'm' and 'M' when the name of the month is desired instead of its number. For example, 'dd-nnn-yyyy' -> '01-Jan-1980'. Uppercase 'N' indicates that the name should be converted to upper case: 'dd-NNN-yyyy' -> '01-JAN-1980'. The abbreviation used is based on the width of the subfield (3 in the examples) and the current contents of the MonthString array.

The characters 'w' and 'W' may be used to include the day of the week in a date string. For example, 'www dd nnn yyyy' -> 'Mon 01 Jan 1989'. Uppercase 'W' indicates that the name should be converted to upper case: 'WWW dd NNN yyyy' -> 'MON 01 JAN 1989'. The abbreviation used is based on the width of the subfield (3 in the examples) and the current contents of the DayString array. (Note that OPFIELD will not allow the user to enter text into a subfield containing 'w' or 'W'. The day of the week will be supplied automatically when a valid date is entered.)

Note that 'Nnn' and 'Www' are not valid components of a picture mask. You *must* use all upper case or all lower case letters: 'nnn', 'NNN', 'www', 'WWW'.

Two of these characters, '/' and ':', are considered "semi-literal." What this means is that when OPDATE sees one of these two characters in a picture mask, it automatically replaces them with the current value of SlashChar (for '/') or ColonChar (for ':'). This allows you, for example, to write a program that contains numerous picture masks like 'mm/dd/yy', for example, and have all of them produce strings in 'mm-dd-yy' format simply by executing the statement "SlashChar := '-';". Alternatively, you could put the '-' in the mask to begin with: 'mm-dd-yy'. Given the correct picture mask and the right settings for ColonChar and SlashChar, OPDATE is thus able to deal with almost any country's conventions for displaying dates and times. (See the documentation for the InternationalDate and InternationalTime routines, as well as the section on "International Currencies and Conventions" earlier in the chapter.)

Any characters not shown in the list above are treated as literals when they appear in a picture mask.

OPDATE does much more than convert dates and times to strings, however. There are several other noteworthy features of the unit, such as the routines for performing date and time arithmetic, as you'll see when you study the remainder of this section.

Valid Range for Dates

OPDATE is designed to allow dates to be stored as either long integers (giving a range of 1/1/1600-12/31/3999) or words (giving a range of 1/1/1900-12/31/2078). The factor that determines which method of storage is used is the conditional compilation directive FourByteDates. By default, FourByteDates is defined, and long integers are used for storage. If you would prefer that dates be stored as words to conserve space, all you need to do is undefine the statement '{$DEFINE FourByteDates}' in OPDEFINE.INC and recompile OPDATE.(Our standard way of doing this is to simply place a period before the $DEFINE.) Among other things, this change will alter the values of several important constants interfaced

by the unit and described later in this chapter: MinYear, MaxYear, MinDate, MaxDate, Date1900, BadDate, and a couple of others. Note that it is assumed throughout the rest of the manual that FourByteDates is defined.

About the Term "Julian Date"

The term "Julian date" means different things to different people. Among astronomers and historians, it has a very precise and technical meaning. Our use of the term will hence probably raise a few eyebrows, because OPDATE doesn't work with "true" Julian dates. Among programmers, however, the term has acquired a much broader meaning, something like "a date stored as an integer value representing the serial number of days from a given point in time." And it is in that widely understood sense that we use the term here.

Important Note to Users of Turbo Professional's TPDATE

In versions of TPDATE prior to 5.09, there was a bug in the code used to calculate Julian dates when the conditional compilation directive FourByteDates was *not* defined. As a result of the bug, any word-sized date greater than or equal to 02/29/2000 (2000 is a leap year) was off by one. Put another way, the formulas used to turn day-month-year into a serial (Julian) date returned the same value for both 02/29/2000 and 03/01/2000.

The bug was fixed in Turbo Professional 5.09, but it has implications for any existing database that contains word-sized dates > = 02/29/2000 that was created using an earlier version of TPDATE. If you are one of the few people whose existing applications fall into that category, you need to write a little utility that will read each record in your database, execute the following statement for each date variable, then write the record back to disk:

```
if DateVar >= 36583 then begin
  Inc(DateVar);
  {... write the record back to disk ...}
end;
```

36583 is the value that was being calculated for both 2/29/2000 and 3/1/2000. The only thing that this will not fix are date variables that were supposed to contain 2/29/2000 to begin with. If feasible, you may want to treat 36583 as a special case, and ask the user to confirm whether a particular date should be 2/29/2000 or 3/1/2000. If it is supposed to be 2/29/2000, the date variable should not be incremented. Keep in mind that such a utility should be run before adding any new records to the database using a program written with OPDATE. The simple "if DateVar > = 36583" test will prove unreliable if the database contains date values calculated both before and after the bug was fixed.

Types

```
{$IFDEF FourByteDates}
Date  = LongInt;
{$ELSE}
Date = Word;
{$ENDIF}
```

In OPDATE, dates are stored in integer format as the number of days since January 1, 1600. Besides offering compact storage–any date through December 31, 3999 may be stored in four bytes–this method facilitates date arithmetic, since the number of days between any two dates may be determined with simple subtraction. Note, however, that if the conditional compilation directive FourByteDates is not defined, dates are stored as words (two bytes), giving a range of 1/1/1900 to 12/31/2078.

```
DateString = string[DateLen];
```

A string representation of a date.

```
DateTimeRec =
  record
    D : Date;
    T : Time;
  end;
```

This record type simply combines the two basic date types defined by OPDATE, Date and Time.

```
DayType = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

An enumerated type used when representing a day of the week.

```
Time = LongInt;
```

OPDATE handles time in a manner similar to dates, representing a given time of day as the number of seconds since midnight. The valid range for a time variable is thus 0 (12 am) to 86,399 (11:59:59 pm), since there are 86,400 (60*60*24) seconds in a day. This method is reasonably compact and facilitates time arithmetic.

Constants

```
{$IFDEF FourByteDates}
BadDate = $FFFFFFFF;
{$ELSE}
BadDate = $FFFF;
{$ENDIF}
```

This value is used to represent an invalid date, such as 12/32/1989.

```
BadTime = $FFFFFFFF;
```

This value is used to represent an invalid time of day, such as 12:61:00.

```
ColonChar : Char = ':';
```

This typed constant determines what character will be used as a separator in time strings. For example, if TimeToTimeString is passed a picture mask of 'hh:mm:ss' and ColonChar is '.', the string returned will be in 'hh.mm.ss' format.

```
{$IFDEF FourByteDates}
Date1900 = $0001AC05;
{$ELSE}
Date1900 = $0000;
{$ENDIF}
```

This constant contains the Julian date for 1/1/1900.

```
{$IFDEF FourByteDates}
Date1980 = $00021E28;
{$ELSE}
Date1980 = $7223;
{$ENDIF}
```

This constant contains the Julian date for 1/1/1980.

```
{$IFDEF FourByteDates}
Date2000 = $00023AB1;
{$ELSE}
Date2000 = $8EAC;
{$ENDIF}
```

This constant contains the Julian date for 1/1/2000.

```
DateLen = 40;
```

This is the maximum length of a Picture string passed to a routine in OPDATE.

```
DayString : array[DayType] of string[15] = (
  'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
  'Saturday');
```

This array can be used to convert DayType variables to strings. For example, if DayOfWeek = Saturday, then the statement 'S := DayString[DayOfWeek]' would set S to 'Saturday'. Notice that you could then get a reasonable abbreviation for the day of the week using 'Copy(S, 1, 3)'. The maximum length for each string is set to 15 to allow the array to be modified for use in foreign countries.

```
HourOnly    = 'h';
MinOnly     = 'm';
SecOnly     = 's';
HourOnlyU   = 'H';
MinOnlyU    = 'M';
SecOnlyU    = 'S';
TimeOnly    = 't';
EmOnly      = 'e';
TimeColon   = ':';
```

These formatting characters are the only ones that are meaningful in a picture mask for a time string. The uppercase versions of the formatting characters cause numbers to be left-padded with spaces (' 1:11:00'), while the lowercase versions cause them to be padded with 0 ('01:11:00'). See the introduction to this chapter for more details.

```
HoursInDay      = 24;
MinutesInHour   = 60;
SecondsInDay    = 86400;
SecondsInHour   = 3600;
SecondsInMinute = 60;
```

These self-explanatory constants are interfaced to facilitate time arithmetic.

```
{$IFDEF FourByteDates}
MinDate = $00000000; {= 01/01/1600}
MaxDate = $000D6025; {= 12/31/3999}
MinYear = 1600;
MaxYear = 3999;
{$ELSE}
MinDate = $0000;     {= 01/01/1900}
MaxDate = $FF62;     {= 12/31/2078}
MinYear = 1900;
MaxYear = 2078;
{$ENDIF}
```

These constants define the valid range of dates for a date variable.

```
MinTime = 0;     {= 00:00:00 am}
MaxTime = 86399; {= 23:59:59 pm}
```

These constants define the valid range of times for a time variable.

```
MonthOnly    = 'm';
DayOnly      = 'd';
YearOnly     = 'y';
NameOnly     = 'n';
WeekDayOnly  = 'w';
MonthOnlyU   = 'M';
DayOnlyU     = 'D';
NameOnlyU    = 'N';
WeekDayOnlyU = 'W';
DateSlash    = '/';
```

These formatting characters are the only ones that are meaningful in a picture mask for a date string. The uppercase versions of the formatting characters cause numbers to be left-

padded with spaces (' 1/11/1900'), while the lowercase versions cause them to be padded with '0' ('01/11/1900'). See the introduction to this chapter for more details.

```
MonthString : array[1..12] of string[12] = (
  'January', 'February', 'March', 'April', 'May', 'June', 'July',
  'August', 'September', 'October', 'November', 'December');
```

This array can be used to convert variables representing months to strings. For example, if Month = 1, then the statement 'S := MonthString[Month]' would set S to 'January'. Notice that you could then get a reasonable abbreviation for the month using 'Copy(S, 1, 3)'. The maximum length for each string is set to 12 to allow the array to be modified for use in foreign countries.

```
SlashChar : Char = '/';
```

This typed constant determines what character will be used as a separator in date strings. For example, if DateToDateString is passed a picture mask of 'mm/dd/yy' and SlashChar is '-', the string returned will be in 'mm-dd-yy' format.

```
Threshold2000 : Integer = 1900;
```

This constant determines how OPDATE interprets the year portion of a date like 01/01/40. If Threshold2000 = 1900 {..1940}, the year is assumed to be 1940. If Threshold2000 = 1941 {..2000}, the year is assumed to be 2040. More generally, Threshold2000 specifies the year at which OPDATE stops assuming a century of 2000 when the year is in the range 00..99 and starts assuming 1900. With the default value of 1900, 1900 is always assumed. With a value of 2000, 2000 would always be assumed.

```
UpCaseTime : Boolean = False; {if true, 't' and 'e' force upper case}
```

This value affects the behavior of routines that return string representations of times. If it is True, and the picture mask passed to the routine contains 't' and/or 'e', the string returned will contain 'P', 'PM', 'A', or 'AM' rather than their lower-case equivalents.

Variables

```
DefaultMonth : Byte;
DefaultYear : Integer;
```

These variables are used by the routines that convert strings to dates in cases where the picture mask represents a partial date such as 'dd' or 'mm/dd'. They are principally useful when OPDATE is being used in conjunction with OPENTRY, and the user is being asked to enter dates that are assumed to fall in a particular month and/or year. They are initialized by OPDATE to the current month and year (assuming that the system clock is accurate).

CurrentTime / CurrentTimeString / DateDiff

Declaration

```
function CurrentTime : Time;
```

Purpose

Returns current time in seconds since midnight.

Description

This function calls DOS to determine the current time of day, then converts the values it returns (hours, minutes, seconds) to a time variable, which is returned as the function result.

Example

```
var Start, Elapsed : Time;
...
Start := CurrentTime;
...
Elapsed := CurrentTime-Start;
if Elapsed < 0 then
  Inc(Elapsed, SecondsInDay);
```

Here CurrentTime is used to calculate the number of seconds that have elapsed between two events.

Declaration

```
function CurrentTimeString(Picture : DateString) : DateString;
```

Purpose

Returns the current time as a string of the specified form.

Description

This function simply calls CurrentTime to get the time of day, then converts it to a string of the form specified by Picture using TimeToTimeString.

Example

```
WriteLn('The time is ', CurrentTimeString('hh:mm:ss te'));
```

Display the current time of day.

See Also

TimeToTimeString

Declaration

```
procedure DateDiff(Date1, Date2 : Date; var Days, Months, Years : Integer);
```

Purpose

Return the difference in Days, Months, Years between two valid Julian dates.

Description

Given two valid Julian dates, Date1 and Date2, this procedure calculates the absolute difference between them in Days, Months, and Years. The order of Date1 and Date2 is not important.

Example

```
var Days, Months, Years : Integer;
...
DateDiff(Today, Today+1, Days, Months, Years);
```

In this case, Days would be 1, while Months and Years would be 0.

See Also

DateTimeDiff TimeDiff

Declaration

```
function DateStringToDate(Picture, S : DateString) : Date;
```

Purpose

Convert the date string S to a Julian date.

Description

Given a string representation of a date (S), this routine converts it to a Julian date, returned as the function result. S must be a string of the same length and form as Picture. If the date represented by S is invalid, BadDate is returned.

Example

```
MyDateString := '01/01/1900';
MyDate := DateStringToDate('mm/dd/yyyy', MyDateString);
```

In this case, MyDate would equal the predefined constant Date1900.

See Also

DateToDateString

Declaration

```
function DateStringToDMY(Picture, S : DateString;
                         var D, M, Y : Integer) : Boolean;
```

Purpose

Extract Day, Month, and Year from S, returning True if string is valid.

Description

Given a string representation of a date (S), this routine extracts the day (D), month (M), and year (Y), and returns True if the date is valid. S must be a string of the same length and form as Picture.

Example

```
DateStringToDate('mm/dd/yyyy', '01/01/1900', D, M, Y);
```

Returns True, with D = 1, M = 1, and Y = 1900.

See Also

DateStringToDate
DMYtoDateString
DateToDateStrings

DateTimeDiff / DateToDateString

Declaration

```
procedure DateTimeDiff(DT1, DT2 : DateTimeRec; var Days : Word;
                       var Secs : LongInt);
```

Purpose

Return the difference in days and seconds between two points in time.

Description

Given two variables of type DateTimeRec, DT1 and DT2, this routine returns the absolute difference between the two points in time in terms of Days and seconds (Secs). The order of DT1 and DT2 is not important.

Examples

```
var
  DT1, DT2 : DateTimeRec;
  Days : Word;
  Secs : LongInt;
...
DT1.D := Today;   DT1.T := CurrentTime;
DT2.D := DT1.D+1; DT2.T := DT1.T+1;
DateTimeDiff(DT1, DT2, Days, Secs);
```

In this case, Days and Secs would both be 1.

```
DT2.T := DT1.T-1;
DateTimeDiff(DT1, DT2, Days, Secs);
```

Days would now be 0 and Secs would be 86,399 (SecondsInDay-1).

See Also

DateDiff TimeDiff

Declaration

```
function DateToDateString(Picture : DateString; Julian : Date) : DateString;
```

Purpose

Converts a Julian to a string of the form indicated by Picture.

Description

Given a date (Julian), this routine converts it to a string of the form specified by Picture and returns it as its function result. Note that if Julian = BadDate, the month, day, and year will be represented as blank spaces.

Examples

```
DateToDateString('mm/dd/yy', MyDateVar);
```

Returns MyDateVar as a string of the form 'mm/dd/yy'.

```
DateToDateString('mm/dd/yy', BadDate);
```

Returns ' / / '.

Declaration
```
procedure DateToDMY(Julian : Date; var Day, Month, Year : Integer);
```

Purpose
Convert from a Julian date to Day, Month, Year.

Description
Given a date (Julian), this routine returns the Day, Month, and Year. If Julian = BadDate, then Day, Month and Year are all set to 0.

Example
```
var Day, Month, Year : Integer;
...
DateToDMY(Today, Day, Month, Year);
```
Sets Day, Month, and Year to today's date.

See Also
DMYtoDate

Declaration
```
function DateToSortString(Julian : Date) : string;
```

Purpose
Convert a date to a sortable string.

Description
Given a Julian date, this routine converts it to a string that can be used by sorting and indexing routines that work only with strings, such as those in our B-Tree Filer and Borland's Database Toolbox. The string produced is 4 bytes long and is not in a printable form.

Example
```
var MySortString : string[4];
...
MySortString := DateToSortString(MyDate);
```
This would convert MyDate to a sortable string.

See Also
SortStringToDate

Declaration
```
function DayOfWeek(Julian : Date) : DayType;
```

Purpose
Return the day of the week for the specified date.

Description
Given a Julian date, this routine returns the corresponding day of the week. Julian is assumed to be a valid date. If it is not, DayOfWeek will return an invalid number (with an ordinal value of 7), prompting a range error if range checking is on.

Example
```
WriteLn('Today is a ', DayString[DayOfWeek(Today)]);
```
Display the day of the week.

See Also
DayOfWeekDMY

DayOfWeekDMY / DaysInMonth / DecTime

Declaration

```
function DayOfWeekDMY(Day, Month, Year : Integer) : DayType;
```

Purpose

Return the day of the week for the day, month, year.

Description

Given a Day, Month, and Year, this routine returns the corresponding day of the week. The date represented by Day, Month, and Year is assumed to be valid. If it is not, DayOfWeek will return an invalid number (with an ordinal value of 7), prompting a range error if range checking is on.

Example

```
DayOfWeek(1, 1, 1990)
```

Returns Monday.

See Also

DayOfWeek

Declaration

```
function DaysInMonth(Month, Year : Integer) : Integer;
```

Purpose

Return the number of days in the specified month of a given year.

Description

Given a valid Month and Year, this function returns the number of days in that month. If the Month is invalid, DaysInMonth will return 0.

Examples

```
DaysInMonth(2, 1990)
```

Returns 28.

```
DaysInMonth(2, 1992)
```

Returns 29, since 1992 is a leap year.

Declaration

```
function DecTime(T : Time; Hours, Minutes, Seconds : Byte) : Time;
```

Purpose

Subtract the specified Hours, Minutes, Seconds from T and return the result.

Description

Given a valid time of day (T), this routine subtracts the specified number of Hours, Minutes, and Seconds from it. The result is adjusted if necessary to account for the rollover at midnight.

Examples

```
RightNow := HMStoTime(2, 0, 0); {02:00:00 am}
HourAgo := DecTime(RightNow, 1, 0, 0);
```

HourAgo would equal 01:00:00 am.

```
ThreeHoursAgo := DecTime(RightNow, 3, 0, 0);
```

ThreeHoursAgo would equal 11:00:00 pm. (Note the rollover.)

See Also

IncTime

Declaration
```
function DMYtoDate(Day, Month, Year : Integer) : Date;
```

Purpose
Convert from Day, Month, Year to a Julian date.

Description
Given a date in the form of a Day, Month, and Year, this routine returns the corresponding Julian date. If the date is invalid, BadDate is returned. If Year is less than 100, it is assumed to be relative to 1900.

Example
```
DateToRemember := DMYtoDate(7, 12, 1941);
```
DateToRemember now holds the number of days between January 1, 1600 and December 7, 1941.

See Also
DateToDMY

Declaration
```
function DMYtoDateString(Picture : DateString;
                         Day, Month, Year : Integer) : DateString;
```

Purpose
Merge the Day, Month, and Year into the Picture.

Description
Given a Day, Month and Year, this routine returns a string representation of the date in the format indicated by Picture. If Year is less than 100, it is assumed to be relative to 1900.

Examples
```
DMYtoDateString('mm/dd/yyyy', 7, 12, 1941);
DMYtoDateString('mm/dd/yyyy', 7, 12, 41);
```
Both return '12/07/1941'.

See Also
DateStringToDMY

Declaration
```
function HMStoTime(Hours, Minutes, Seconds : Byte) : Time;
```

Purpose
Convert Hours, Minutes, Seconds to a Time variable.

Description
Given a time of day in terms of Hours, Minutes, and Seconds, this routine returns the time in seconds since midnight. Hours is assumed to be in military format (0-23). If the time is invalid, BadTime is returned.

Example
```
HMStoTime(1, 2, 3);
```
Returns 3723–(1*SecondsInHour)+(2*MinutesInHour)+3–or 01:02:03 am.

See Also
TimeToHMS

IncDate / IncDateTime

Declaration

```
function IncDate(Julian : Date; Days, Months, Years : Integer) : Date;
```

Purpose

Add (or subtract) the specified number of Days, Months, and Years to a date.

Description

Given a valid Julian date, this routine adjusts it by the specified number of Days, Months, and Years. Months and Years are added before days, and no overflow/underflow checks are made because the result is assumed to be a date in the range MinDate..MaxDate.

Note that there is one tricky case involving IncDate, the case where Julian represents a date at the end of a month and the addition of one or more months can result in a day of the month that doesn't exist in the new month. For example:

```
D1 := DMYtoDate(29, 1, 1990); {01/29/1990}
D2 := IncDate(D1, 0, 1, 0);
```

The result "should" be 02/29/1990, but that date doesn't exist. So what you get is a seemingly illogical result that is consistent with IncDate's simplistic philosophy of date arithmetic: 03/01/1990. That is, the value in D2 would represent 02/29/1990 if there was such a date, but because February has only 28 days in that year, the result is Date(02/28/1990)+1, or 03/01/1990. If this behavior is unacceptable for your application, use IncDateTrunc instead.

Examples

```
YearFromNow := IncDate(Today, 0, 0, 1);
```

YearFromNow would hold the Julian representation of the date 1 year from today.

```
YearAgo := IncDate(Today, 0, 0, -1);
```

YearAgo would hold the Julian representation of the date 1 year ago.

See Also

IncDateTrunc

Declaration

```
procedure IncDateTime(var DT1, DT2 : DateTimeRec;
                      Days : Integer; Secs : LongInt);
```

Purpose

Increment (or decrement) DT1 by the specified number of days and seconds and put the result in DT2.

Description

Given a valid DateTimeRec, DT1, this routine adjusts its date and time by the specified number of Days and seconds, then stores the result in DT2. (DT1 is unchanged.)

Examples

```
var
  StartTime, StopTime : DateTimeRec;
...
IncDateTime(StartTime, StopTime, 0, 8*SecondsInHour);
```

StopTime will have the date and time 8 hours from StartTime.

```
IncDateTime(StartTime, StartTime, -7, 0);
```

StartTime will have the date and time one week ago.

See Also

IncTime

Declaration

```
function IncDateTrunc(Julian : Date; Months, Years : Integer) : Date;
```

Purpose

Add (or subtract) the specified number of months and years to a date.

Description

This routine is essentially the same as IncDate, but with two important differences. First, it takes no Days parameter, just Months and Years. And second, it "truncates" the day of the month in cases where the addition of Months results in an invalid date. See the example below for an illustration of what is meant by "truncates."

Example

```
D1 := DMYtoDate(29, 1, 1990); {01/29/1990}
D2 := IncDate(D1, 0, 1, 0);
D3 := IncDateTrunc(D1, 1, 0);
```

Because 02/29/1990 does not exist, D2 would be set to 03/01/1990 and D3 would be set to 02/28/1990.

See Also

IncDate

Declaration

```
function IncTime(T : Time; Hours, Minutes, Seconds : Byte) : Time;
```

Purpose

Add the specified Hours, Minutes, and Seconds to T and return the result.

Description

Given a time of day (T), this routine adds the specified number of Hours, Minutes, and Seconds to it. The result is adjusted if necessary to account for the rollover at midnight.

Examples

```
RightNow := HMStoTime(22, 0, 0); {10:00:00 pm}
HourFromNow := IncTime(RightNow, 1, 0, 0);
```

HourFromNow would equal 11:00:00 pm.

```
ThreeHoursFromNow := IncTime(RightNow, 3, 0, 0);
```

ThreeHoursFromNow would equal 1:00:00 am. (Note the rollover.)

See Also

DecTime

InternationalDate / InternationalTime

Declaration

```
function InternationalDate(WholeYear, ZeroPad : Boolean) : DateString;
```

Purpose

Return a picture mask for a date string, based on DOS's country info.

Description

This routine uses DOS's Get Country Information service in order to determine not only the proper value for SlashChar, but also how to produce a picture mask that is appropriate for the current country.

The WholeYear parameter indicates whether the picture mask should leave room for a century ('yyyy') as well as a year ('yy'). ZeroPad indicates whether the first subfield in the picture mask should be padded with zeroes or with spaces.

Note that the information provided by DOS 2.x is minimal compared to that available under DOS 3.x and higher. We therefore recommend that this function be used with caution in programs that must run under DOS 2.

Examples

```
InternationalDate(True, True)
```

Returns 'mm/dd/yyyy' for US (SlashChar = '-'), 'yyyy/mm/dd' for Sweden (SlashChar = '-').

```
InternationalDate(False, False)
```

Returns 'MM/dd/yy' for US, 'DD/mm/yy' for Switzerland (SlashChar = '.').

See Also

InternationalCurrency (OPABSFLD)

Declaration

```
function InternationalTime(WithSeconds, ZeroPad : Boolean;
                           ExtraSpace, WithEm : Boolean) : DateString;
```

Purpose

Return a picture mask for a time string, based on DOS's country info.

Description

This routine uses DOS's Get Country Information service in order to determine not only the proper value for ColonChar, but also how to produce a picture mask that is appropriate for the current country.

WithSeconds indicates whether the mask should make room for seconds as well as hours and minutes. ZeroPad indicates whether the hours subfield should be padded with zeroes or spaces. WithEm indicates whether or not 'te' should be added into the end of the string if appropriate. And, if 'te' is appended, ExtraSpace indicates whether or not there should be a space in front of it.

Note that DOS 2.x provides no information about how time strings should be formatted. (DOS versions numbered 3.0 and higher do.) We therefore recommend that this function not be used in programs that must run under DOS 2.

Examples

```
InternationalTime(True, True, True, True)
```

Returns 'HH:mm:ss te' for US (ColonChar = ':'), 'HH:mm:ss' for Norway (ColonChar = '.').

```
InternationalTime(False, False, False, False)
```

Returns 'hh:mm' for US, 'hh:mm' for Denmark (ColonChar = '.').

See Also

InternationalCurrency (OPABSFLD)

Declaration

```
function IsLeapYear(Year : Integer) : Boolean;
```

Purpose

Return True if Year is a leap year.

Description

This function will return True if the specified Year is a leap year. No error checking is done by IsLeapYear; Year is assumed to be in the range MinYear..MaxYear.

Example

```
for Year := 1900 to 2000 do
  if IsLeapYear(Year) then
    WriteLn(Year);
```

Display all leap years from 1900 to 2000, inclusive.

Declaration

```
function RoundToNearestHour(T : Time; Truncate : Boolean) : Time;
```

Purpose

Round T to the nearest hour, or Truncate minutes and seconds from T.

Description

This routine does one of two things, depending on the value of Truncate. If Truncate is True, it simply sets the minutes and seconds in T to 0 and returns the value (e.g., 4:01:02 becomes 4:00:00). If Truncate is False, it rounds the time to the nearest hour (e.g., 4:30:00 becomes 5:00:00). T is assumed to be a valid Time variable.

See Also

RoundToNearestMinute

RoundToNearestMinute / SortStringToDate

Declaration

```
function RoundToNearestMinute(T : Time; Truncate : Boolean) : Time;
```

Purpose

Round T to the nearest minute, or Truncate seconds from T.

Description

This routine does one of two things, depending on the value of Truncate. If Truncate is True, it simply sets the seconds in T to 0 and returns the value (e.g., 4:01:02 becomes 4:01:00). If Truncate is False, it rounds the time to the nearest minute (e.g., 4:00:30 becomes 4:01:00). T is assumed to be a valid Time variable.

See Also

RoundToNearestHour

Declaration

```
function SortStringToDate(S : string) : Date;
```

Purpose

Convert a sortable string to a date.

Description

Given a sortable string (S) produced by DateToSortString, this routine converts it back to a Julian date, which is returned as the function result.

Example

```
var MySortString : string[4];
    MyDate : Date;
...
MySortString := DateToSortString(MyDate);
...
MyDate := SortStringToDate(MySortString);
```

This would convert MyDate to a sortable string and then back again.

See Also

DateToSortString

Declaration
```
function SortStringToTime(S : string) : Time;
```

Purpose
Convert a sortable string to a time variable.

Description
Given a sortable string (S) produced by TimeToSortString, this routine converts it back to a time variable, which is returned as the function result.

Example
```
var MySortString : string[4];
    MyTime : Time;
...
MySortString := TimeToSortString(MyTime);
...
MyTime := SortStringToTime(MySortString);
```
This would convert MyTime to a sortable string and then back again.

See Also
TimeToSortString

Declaration
```
procedure TimeDiff(Time1, Time2 : Time; var Hours, Minutes, Seconds : Byte);
```

Purpose
Return the difference in Hours, Minutes, and Seconds between two times.

Description
Given two valid time variables, Time1 and Time2, this procedure calculates the absolute difference between them in Hours, Minutes, and Seconds. The order of Time1 and Time2 is not important.

Example
```
var
  Time1, Time2 : Time;
  Hours, Minutes, Seconds : Byte;
...
Time1 := CurrentTime;
Time2 := IncTime(Time1, 1, 2, 3);
TimeDiff(Time1, Time2, Hours, Minutes, Seconds);
```
In this case, Hours would be 1, Minutes would be 2, and Seconds would be 3.

See Also
DateDiff DateTimeDiff

TimeStringToHMS / TimeStringToTime

Declaration

```
function TimeStringToHMS(Picture, St : DateString;
                         var H, M, S : Integer) : Boolean;
```

Purpose

Extract Hours, Minutes, Seconds from St, returning True if string is valid.

Description

Given a string representation of a time (St), this routine converts it to hours (H), minutes (M) and seconds (S). St must be a string of the same length and form as Picture. If the time represented by St is invalid, the function returns False.

Example

```
TimeStringToHMS('hh:mm:ss', '01:02:03', H, M, S)
```

Returns True, with H = 1, M = 2, and S = 3.

Declaration

```
function TimeStringToTime(Picture, S : DateString) : Time;
```

Purpose

Convert S, a string of the form indicated by Picture, to Time.

Description

Given a string representation of a time (S), this routine converts it to a time variable, returned as the function result. S must be a string of the same length and form as Picture. If the time represented by S is invalid, BadTime is returned.

Example

```
MyTimeVar := TimeStringToTime('hh:mm:ss', '01:02:03');
```

MyTimeVar would be set to 1:02:03 am.

See Also

TimeToTimeString TimeToAmPmString

Declaration

```
function TimeToAmPmString(Picture : DateString; T : Time) : DateString;
```

Purpose

Convert T to a string of the form indicated by Picture. Times are always represented in am/pm format.

Description

This routine is identical to TimeToTimeString except that it forces the time designated by T to be represented in am/pm format, even if the Picture mask does not require it. Note that if Time = BadTime, the hour, minutes, and seconds will be represented as blank spaces.

Examples

```
MyTimeString1 := TimeToAmPmString('hh:mm:ss', 0);
MyTimeString2 := TimeToTimeString('hh:mm:ss', 0);
MyTimeString3 := TimeToAmPmString('hh:mm:ss', BadTime);
```

MyTimeString1 would be set to '12:00:00', MyTimeString2 to '00:00:00', and MyTimeString3 to ' : : '.

See Also

TimeToTimeString

Declaration

```
procedure TimeToHMS(T : Time; var Hours, Minutes, Seconds : Byte);
```

Purpose

Convert a Time variable to Hours, Minutes, and Seconds.

Description

Given a time variable (T), this routine extracts the Hours, Minutes, and Seconds. If Time = BadTime, Hours, Minutes, and Seconds are all set to 0.

Example

```
var Hours, Minutes, Seconds : Byte;
...
TimeToHMS(CurrentTime, Hours, Minutes, Seconds);
```

Sets Hours, Minutes, and Seconds to the current time.

See Also

HMStoTime

TimeToSortString / TimeToTimeString / Today

Declaration

```
function TimeToSortString(T : Time) : string;
```

Purpose

Convert a time variable to a sortable string.

Description

Given a time variable (T), this routine converts it to a string that can be used by sorting and indexing routines that work only with strings, such as those in our B-Tree Filer and Borland's Database Toolbox. The string produced is 4 bytes long and is not in a printable form.

Example

```
var MySortString : string[4];
...
MySortString := TimeToSortString(MyTime);
```

This would convert MyTime to a sortable string.

See Also

SortStringToTime

Declaration

```
function TimeToTimeString(Picture : DateString; T : Time) : DateString;
```

Purpose

Convert T to a string of the form indicated by Picture.

Description

Given a time of day (T), this routine converts it to a string of the form specified by Picture and returns it as its function result. Note that if Time = BadTime, the hour, minutes, and seconds will be represented as blank spaces.

Examples

```
TimeToTimeString('hh:mm:ss', MyTimeVar);
```

Returns MyTimeVar as a string of the form 'hh:mm:ss'.

```
TimeToTimeString('hh:mm:ss', BadTime);
```

Returns ' : : '.

Declaration

```
function Today : Date;
```

Purpose

Returns today's date as a Julian.

Description

This function calls DOS to determine the current date, then converts the values it returns (day, month, year) to a date variable, which is returned as the function result.

Example

```
WriteLn('Today is a ', DayString[DayOfWeek(Today)]);
```

Display the day of the week.

See Also

DayOfWeek TodayString

Declaration
```
function TodayString(Picture : DateString) : DateString;
```

Purpose
Returns today's date as a string of the specified form.

Description
This function simply calls Today to get the current date, then converts it to a string of the form specified by Picture using DateToDateString.

Example
```
WriteLn('Today is ', TodayString('mm/dd/yyyy'));
```
Display today's date.

See Also
DateToDateString Today

Declaration
```
function ValidDate(Day, Month, Year : Integer) : Boolean;
```

Purpose
Verify that Day, Month, Year is a valid date.

Description
This routine is intended primarily for internal use, but you may use it if you wish. Given a Day, Month, and Year, it returns True if the date is valid and False if it isn't. If Year is less than 100, it is assumed to be relative to 1900. If Year is not less than 100, it must be in the range 1600-3999.

Examples
```
ValidDate(1, 1, 1598)
```
Would return False because the year is out of range.
```
ValidDate(4, 31, 1988)
```
Would return False because there are only 30 days in April.

Declaration
```
function ValidTime(Hours, Minutes, Seconds : Integer) : Boolean;
```

Purpose
Return True if Hours:Minutes:Seconds is a valid time.

Description
This routine is intended primarily for internal use, but you may use it if you wish. Given a time of day represented in terms of Hours, Minutes, and Seconds, it returns True if the time is valid and False if it isn't.

Examples
```
ValidTime(24, 0, 0)
```
Would return False because Hours should be in the range 0..23.
```
ValidTime(0, 0, 61)
```
Would return False because Seconds should be in the range 0..59.

J. Utility and Sample Programs

This section covers three programs: MAKESCRN, a utility for designing data entry screens interactively; ENTRY, a comprehensive example of how to use the EntryScreen object; and BEST, a program that demonstrates the use of the ScrollingEntryScreen.

1. MAKESCRN

MAKESCRN does for OPENTRY what MAKEMENU does for OPMENU. That is, it lets you design your data entry screens interactively, save them in libraries, and generate source code that can be easily integrated into your own programs.

If you are going to use OPENTRY at all, you will definitely want to learn how to use this utility. It can make the task of designing data entry screens significantly easier, and it has the potential to save you several hours of work every time you use it. The time you invest in learning to use it will be well spent, we assure you.

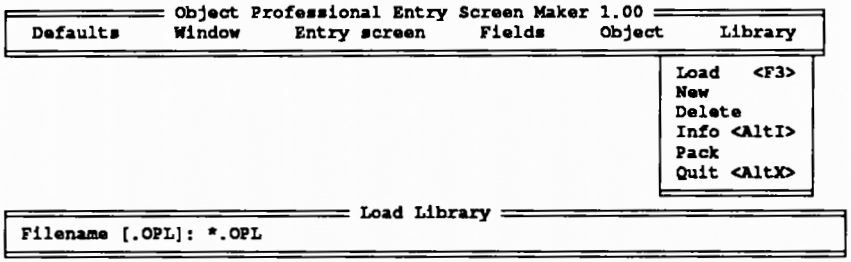
Before forging ahead, please be sure that you have read at least the introduction to the documentation for OPENTRY, as well as the relevant portions of the documentation for OPABSFLD, OPFIELD, OPFEDIT, and OPWINDOW. By "relevant portions," we mean those sections that pertain directly and indirectly to the EntryScreen object. We will assume, for example, that you are already familiar with picture masks, fields, field options, field editors, windows, borders, and so on. Some familiarity with the concept of object libraries (implemented in OPROOT) will also be helpful.

Getting Started

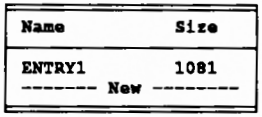
To introduce you to MAKESCRN, we'll start by looking at a sample entry screen that we've prepared for this purpose. For this exercise you'll need two files: MAKESCRN.EXE and SAMPLE.OPL. Assuming that they are both in the current directory, simply enter

```
MAKESCRN
```

at the command line. What you'll see is something like this:

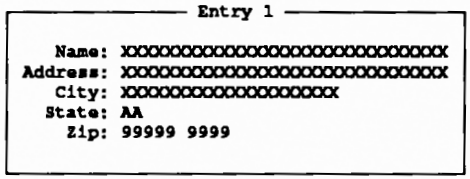


The Load item on the Library submenu has been pre-selected for you, and all you need to do now is enter the name of a new or existing library ("OPL" means **Object Professional** Library). We could enter "SAMPLE" at this prompt–a file extension of "OPL" is assumed–but instead let's just press <Enter>. This will bring up a directory list showing all the files with an extension of "OPL" in the current directory. Unless you have other OPL files in this directory, your only choice should be "sample.opl," which you can now select by pressing <Enter> again. Now you'll be shown a list of all the entry screens in SAMPLE.OPL. There may be objects of other types in the library, and in this case there are, but you'll only see the names of the entry screens. The list should look something like this:



You have two choices here. You can create a new entry screen by selecting "New", or you can load the existing object ENTRY1. Press <Enter> to do so.

When you select ENTRY1 from the list, the object will be displayed in the center of the screen, with the backdrop (the block characters with the polka-dots form the default backdrop) underneath it and the main menu on top of it. Press <F10> to leave the menu system and enter Edit mode. What you'll see now is something like this:



As you can see, this particular entry screen has five fields. The words on the left ('Name:', 'Address:', etc.) are the prompts for the fields. The strings of characters to their right ('AA', '99999 9999', etc.) represent the picture masks for the fields.

At the bottom of the screen is a status line, which is divided into two parts:

```
<Enter> Edit  <AltT> Test  <F2> Save  <F10> Menu | ENTRY1          Row 1   Col 1
```

On the left are the key assignments for some commonly used functions--more on them in a moment. On the right are the name of the entry screen and the current position of the cursor. This status line can be positioned at the top or bottom of the screen, or you can get rid of it entirely. To cycle through the three choices, press <AltS> (for Status) repeatedly.

Before trying to modify the entry screen, let's try testing it to see how it would look and act in an actual program. To do so, press <AltT> (for Test). At this point the picture masks will disappear and you'll see a blank entry screen that is ready to have data entered into it. The cursor should be on the Name field. If you left the status line at the bottom of the screen, you should see a slightly different one there now:

```
<Esc> Exit  <^Enter> Done  <AltS> Toggle status  | ENTRY1          Name
```

On the left are a new set of reminders about key assignments. On the right are the name of the entry screen, as before, and the symbolic name of the current field. If you press <Enter> repeatedly, you'll see that this field name will change as you move from field to field. Notice that the <AltS> command works here as well.

This Test mode gives you an opportunity to try out your entry screen and determine how it will work when you plug it into your own program. Although there's no point in spending much time testing this particular entry screen, you should at least try entering your own name in the Name field at the top. Don't press either of the shift keys--just enter your name in lower case letters. You may fill out the rest of the fields too if you wish.

Now we're ready to try modifying the entry screen a bit. To get out of Test mode and return to Edit mode, press <Esc> or <CtrlEnter>. Using the cursor keys, move the cursor until it is on top of some portion of the first field, either the prompt ('Name:') or the picture mask. Now

press <Enter> to edit the field definition. What you'll see is another entry screen that looks something like this:

```
                    Name
Name        Name_____________
                   Prompt
Prompt      Name:
Row, Col    2   5
                    Field
Type        String
Editor      Normal
Picture     xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Row, Col    2   11
Width       30
Pad char    ' '
Options
                    Range


          Normal ---- Selected -- Protected -
        Color Mono   Color Mono   Color Mono
Prompt    1F   07      1F   07      17   07
Field     1E   07      31   0F      17   07
Control   1E   0F
                  Instructions
      Enter symbolic Pascal name for field
```

We'll discuss each of the fields in this entry screen later. For now, we'll just modify one of them. Press the <Down> arrow until the cursor is on the Picture field. What we want to do here is to change the picture mask from 30 X's (upper case) to 30 x's (lower case). We could do that by typing them in one at a time, but in this case we'll use a shortcut. Press <F10>, which will open a popup window prompting you for a picture mask character. Ignore the default ('X') and enter 'x'. You'll then be asked for the length of the picture mask. Press <Enter> to accept the default response of 30. The Picture field should now contain a string 30 characters long consisting of x's. Press <CtrlEnter> to indicate that you are through modifying the field.

We're back in Edit mode now, and the Name field should reflect the change we just made. To see what effect it had, press <AltT> to go back into Test mode. The values you entered earlier will be gone now--they get thrown away as soon as you add, copy, delete, or edit a field in the entry screen. But that's just as well, because we want to reenter your name in the first field. Again, don't press either of the shift keys while typing--just enter your name in lower case. This time, however, your name will appear in mixed case: if you enter "john doe", it will come out "John Doe". This is because the picture mask character 'x' means "force mixed case during data entry." Press <Esc> to return to Edit mode.

Now let's try adding another field, a Phone number field. Move the cursor to row 7, column 1, the start of the row beneath the Zip field. Press <F10> to activate the main menu, then select Fields-Add-String. This will bring up a blank entry screen just like the one we saw earlier. We only need to fill in a few fields here to do what we want. In the Name field, enter 'Phone' and press <Enter>; this gives our field a symbolic name. In the Prompt field, enter 'Phone:' and press <Enter>. You'll notice that the column coordinate for the field has changed automatically from 1 to 8, which is where the field would start if there was to be one space between the end of the prompt and the start of the field. (The column coordinate is changed automatically like this whenever the picture mask is blank or an edit made to the Prompt causes it to overlap the field.)

Now press <Down> or <Enter> until you get to the Picture field, enter '(999) 999-9999', and press <Enter>. You'll notice that the field width has also been changed automatically, from 1

to 14, the length of our picture mask. (The field width, shown only for string and array of char fields, is updated automatically whenever the length of the picture mask exceeds the width.) The defaults for the rest of the fields are fine, so all we have to do now is press <CtrlEnter> to accept our changes and return to the main menu. Now press <F10> to return to Edit mode.

Unfortunately, our new field isn't aligned properly with the others. We could press <Enter> here to edit the field definition, but let's do it another way. Press <F10> to get back into the menu and select Fields-Move. The menu will disappear, and the cursor (now a blinking block cursor) will be positioned at the start of the prompt for the field. We need to move the whole field three columns to the right, so that the 'P' in 'Phone' is beneath the 'S' in State. So press <Right> three times to move the cursor to row 7, column 4 and press <Enter>. Our field should now be aligned properly. Press <F10> to leave the menu and return to Edit mode.

Now we can see a new problem: there is still a blank line at the top of the entry screen, but there is no longer one at the bottom. To fix this, we need to resize the window, so press <AltR> (for Resize). You should see a single-line box that represents the dimensions for the inner part of the window, and the cursor should be at the bottom right corner of this box. If the status line is still visible, it should look something like this:

```
<Enter> Accept <ScrollLock> Toggle | Resize  Width 41  Height 7  Row 15  Col 60
```

On the right side of the status line, we see that the width of the inner window is 41, its height is 7, and the bottom right corner is at row 15, column 60. We need to increase the height by 1, so press <Down> to expand the box by one row.

This is all we set out to do, so we could just press <Enter> to accept our changes. But our entry screen was centered in the middle of the screen before we changed its size, and we want to be sure that it is centered again. So press <ScrollLock> to toggle between Resize and Move modes (the status line will show 'Move' just to the right of the divider). The cursor will now jump to the top left corner, and we can drag the whole window around the screen by pressing the cursor keys. Try pressing <Left> and <Up> until the cursor is at, say, row 3, column 5.

This is not what we want, of course–our window isn't even close to being centered. No problem. Press <AltH> to center the entire window Horizontally, then <AltV> to center it Vertically. This is where we want it. Nevertheless, let's try one more thing. Use the cursor keys to move the window off-center again. This time, press <AltC> to Center the window both horizontally *and* vertically. We should point out here that <AltH>, <AltV>, and <AltC> work in both Move and Resize modes. We switched into Move mode only to show you how it works. Press <Enter> to accept the changes we just made.

Those are all the changes we're going to make to this entry screen. We could save them right now by pressing <F2>, but that would make them permanent, and you might want to repeat this lesson. So instead let's save the modified entry screen under a new name. Press <F10> to go into the menu system and select Object-Save as. At the prompt for a name, enter "ENTRY2". You'll see a message saying that the object has been saved, and, if you look down at the status line you'll see that the name there has changed from "ENTRY1" to "ENTRY2".

We're not quite done with this entry screen, though, because we don't yet have anything we can use in a program. From the menu system, select Object-Write source. You'll then see a popup window asking you for a filename:

```
Name of source file [.PAS]: ENTRY2.PAS
```

Press <Enter> to select the default response. In a few seconds, you'll see a message indicating that your program has been generated. The result is a ready-to-run test program. To verify that

it works, press <AltX> to exit from MAKESCRN, then use either compiler (TURBO or TPC, preferably the latter) to compile it to an EXE file. You'll probably want to glance at the source code that was generated at this point, but that's up to you. We'll talk more about source code generation later.

Your First Entry Screen

As you've seen, modifying an existing entry screen is quite easy. Creating a new one isn't really much harder, though, and that's what we'll be doing in this section.

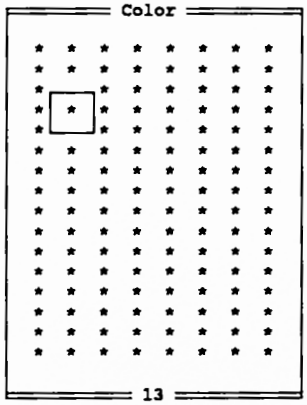
The first step is to load a library. We've already learned how to load an existing library, so let's create a new one instead. When MAKESCRN starts, and you're asked for the name of a library, don't press <Enter> to get a directory listing. Enter a new name instead, say "MYLIB". MAKESCRN won't be able to find MYLIB.OPL, of course, so it will ask if you want to create it. Press <Y> or <Enter>. You'll then be asked for the number of directory entries in the library. This value determines how many objects can be stored in it. To accept the default response of 15, press <Enter>.

As before, you'll now see a list of entry screens to load. This time, however, the list will be empty, and your only choices will be to press <Esc> to cancel the operation or press <Enter> to indicate that you want to create a new one. When you press <Enter>, you'll be asked for a name for your entry screen. Be careful here. Although you are allowed to enter a string of up to 12 characters, you will probably never want to enter one that long. The name specified here will be used as the name of your program when source code is generated, so it is wise to enter a string of no more than 8 characters that both DOS and the compiler will be happy with.

After you've entered a name, you'll see an empty window near the top left corner of the screen, waiting to be resized and moved into position. We do this the same way that we resized the other entry screen earlier. We want to make the entry screen much bigger initially, so press <CtrlRight> until the status line shows the width as 45. We could do this with <Right>, but it would take 8 times as many keystrokes. Now press <PgDn> to increase the height to 11. Again, we could do this with <Down>, but it would take 5 keystrokes instead of 1. Press <AltC> to center the window on the screen, then <Enter>. We'll adjust the size later when we have a better idea of how big it really needs to be.

Next we'll add some enhancements to the window. From the main menu, select Window-Border. This brings up a list of built-in border types to choose from. If you move the cursor up and down the list, you'll see that the top of the pick list displays an example of the currently highlighted item. Select any border type you want except '--none--'; we suggest the double-line border, second from the top.

Now let's change the border's color. From the menu, select Window-Colors-Border. You should see something like this:



Using the cursor keys, move the little box to the bottom of the same column. The value at the bottom of the border will be 1F (hex), which corresponds to White on Blue. This is the attribute to be used on color systems (note the header at the top). Press <Enter> to accept the new attribute. Now you'll be asked to select the attribute to be used on monochrome systems (note the "Mono" at the top). Again, move the little box to the bottom of the same column, so that 0F (White on Black) is displayed at the bottom of the border. Press <Enter> to accept the new setting. The border around the window, partially visible beneath the menu, should now be displayed in the new color.

Next we'll add a "special effect." Select Window-Effects from the menu, then move the cursor to the Shadows item. This menu choice lets you toggle between three settings: None (no shadows), Big, and Small. Try pressing <Enter> repeatedly to see what difference each setting makes. Stop when you get to 'Big' the second time.

Finally, we'll add a header to the window, similar to the one on our first entry screen. From the menu, select Window-Headers-Add-Top center. When prompted for the header string, enter ' Entry 3 '. You'll then be asked to set the attributes for the header. Press <Enter> twice to accept the defaults.

Now we're ready to add a couple of fields, so press <F10> to go into Edit mode. Move the cursor to row 2, column 2, then press <F10> and select Fields-Add-String. In the Name field, enter 'Name'. In the Prompt field, enter 'Name:'. And in the Picture field enter a string consisting of 30 X's using the <F10> shortcut. Press <CtrlEnter> to accept the remaining defaults.

Next move the cursor to row 3, column 3 and press <F10> to return to the Fields-Add submenu. Select LongInt from this menu. In the Name field, enter 'Age'. In the Prompt field, enter 'Age:'. In the Picture field, enter '999'. If you want, you might enter some reasonable Lower and Upper Limits here, say 1 and 110. Hmmm. We don't really need a LongInt to store numbers this small, so let's move the cursor to the Type field and press <Space> or < + > until 'Byte' appears instead of 'LongInt'. Press <CtrlEnter> to accept the remaining defaults.

At this point, we decide that we want to add another field in between the two we've just added. Before we can add it, we need to move the Age field down a row. The fastest way to do this is to use the insert line command. Move the cursor to row 3, column 2 and press <CtrlN>. Now we can add another field between the two we have. Press <F10> and select Fields-Add-Date. In the Name field, enter 'Born'. In the Prompt field, enter 'Born:'. Move the cursor to the

Picture field, press <End> and add 'yy' to the end of it, giving us 'mm/dd/yyyy'. Press <CtrlEnter> to accept the remaining defaults.

Now that we know just how big our entry screen needs to be, we can adjust its size. Press <AltR> and use the <Left> and <Up> keys to adjust the width to 38 and the height to 5. Press <AltC> to center it, then <Enter> to accept the changes.

Everything looks fine, so we can test what we have. Press <AltT> to enter Test mode. When you're finished experimenting with it, press <Esc> or <CtrlEnter> to return to Edit mode. Let's try one more thing. Move the cursor to row 4, where the Age field is, and press <CtrlN> four times. The Age field will be moved out of view. Now press <AltT> again. As you move the cursor from field to field, you'll see that the window scrolls when the cursor goes in or out of the Age field. This is all you have to do to create a scrollable entry screen: add a field that is outside the window. When it comes time to generate source code, MAKESCRN will see that all the fields can't fit inside the window at once, so it will generate code for a ScrollingEntryScreen rather than a regular EntryScreen. Press <Esc> to return to Edit mode, and move the cursor back to row 4. Press <CtrlY> four times to delete the lines we inserted earlier.

Only two things remain to be done. First, press <F2> to save the entry screen we just created. Second, go into the main menu and select Object-Write source to generate the source code.

That's all there is to it. You've just finished your first entry screen, and we've just finished our little tutorial. There are several other commands and menu choices you need to learn about, of course, but the basic ones are behind you. With a little experimentation, you should be able to figure out how to use the other features of the program on your own after reading the remainder of this section.

The Menu System

When an object is loaded, <F10> and <ClickBoth> act as hotkeys that can get you into or out of the menu system immediately. <Esc> and <ClickRight> do much the same thing, but when you are in the menu system they take you to the next highest submenu; only when the topmost menu is active do they take you back to Edit mode.

When an object is not loaded, <F10> and <ClickBoth> do nothing. If a submenu is active, <Esc> and <ClickRight> take you to the next highest submenu. If the topmost menu is active, they can be used to exit from the program.

What follows is a summary of all the choices in the main menu. If there is a hotkey that corresponds to a given menu choice, it is displayed to the right of the name of the menu choice. All of the hotkeys shown here may be used while in Edit mode as well as in the menu system.

Defaults

Backdrop
Lets you select from one of three background types: polka-dot blocks (default), ruler lines, or solid black.

Status line **<AltS>**
Toggles the status line, which can appear at the bottom of the screen (default), top of the screen, or not at all.

Color constants
This option setting affects the way source code is generated. If it is On ('Yes'), all video attributes are represented using the constant names defined in OPCOLOR (e.g.,

'WhiteOnBlack'), and OPCOLOR appears in the USES list. If it is Off ('No'), hexadecimal numbers (such as $0F) are used instead.

Help constants
This option determines whether or not the source code generated contains help index constants ('hiField000') as well as field constants ('idField000').

User proc hooks
This option determines whether or not the source code generated includes shells for user-written procedures of the following types: pre-edit, post-edit, IncChoice, error handler, and context-sensitive help.

Window

Border
Select a window border.

Colors
These menu choices allow you to set various video attributes. Where appropriate, you will be asked if you want the change to be applied globally to existing fields. (Note that the Text setting applies to text fields, as well as to the background color for the window.) If you answer No, the new setting will simply become the default setting for fields added thereafter. The list below shows the name of the menu choice, followed by the names of the corresponding fields in a ColorSet (see the documentation for OPCRT):

Text	TextColor	TextMono
Border	FrameColor	FrameMono
Header	HeaderColor	HeaderMono
Shadow	ShadowColor	ShadowMono
Scroll bar	ScrollBarColor	ScrollBarMono
Slider	SliderColor	SliderMono
Hot spot	HotSpotColor	HotSpotMono
Mouse	MouseColor	MouseMono
Prompt	PromptColor	PromptMono
Selected prompt	SelPromptColor	SelPromptMono
Protected prompt	ProPromptColor	ProPromptMono
Field	FieldColor	FieldMono
Selected field	SelFieldColor	SelFieldMono
Protected field	ProFieldColor	ProFieldMono
Control characters	CtrlColor	CtrlMono
Delimiter	DelimColor	DelimMono
Selected delimiter	SelDelimColor	SelDelimMono
Protected delimiter	ProDelimColor	ProDelimMono

Effects

Delay
Prompts for the delay (in milliseconds) between each stage of an explosion.

Explosions
Toggles between exploding and non-exploding windows.

Shadows
Toggles between Big shadows, Small shadows, and no shadows (None).

Sound
Toggles sound effects (during explosions) on and off.

Headers

Add
Displays a submenu showing the 6 possible positions for the header. Once the position is selected you will be prompted for the header itself, and then given a chance to set its display colors.

Edit
Displays a pick list of existing headers. Move the cursor to the header you want to edit and press <Enter>. You can change both the text of the header and its colors.

Remove
Displays a pick list of existing headers. Move the cursor to the header you want to remove and press <Enter>.

Move **<AltM>**
Move the entry screen.

Resize **<AltR>**
Resize and/or move the entry screen.

ScrollBars

Bottom
Toggle scroll bar at bottom edge of border.

Right
Toggle scroll bar at right edge of border.

Slider char
Set character to use for slider.

Bar char
Set character to use for scroll bar.

Arrow top
Set character to use for up arrow on vertical scroll bar.

Arrow bottom
Set character to use for down arrow on vertical scroll bar.

Arrow left
Set character to use for left arrow on horizontal scroll bar.

Arrow right
Set character to use for right arrow on horizontal scroll bar.

Entry screen

Field options
These menu choices allow you to activate/deactivate various field options. Where appropriate, you will be asked if you want the change to be applied globally to existing fields. If you answer No, the new setting will simply become the default setting for new fields. The list below shows the name of the menu choice, followed by the corresponding field option (see the OPFIELD section earlier in this chapter). The Pad character only affects the way that a field is displayed; it does not affect the actual contents of the field.

```
Auto advance char       efAutoAdvanceChar
Auto advance cursor     efAutoAdvanceCursor
Auto NumLock            efAutoNumLock
Beep on error           efBeepOnError
Clear first char        efClearFirstChar
Cursor to end           efCursorToEnd
Exit on mouse click     efClickExit
Force case              efForceMode
Force typing mode       efForceOvertype
House cursor at end     efHouseCursorAtEnd
Insert pushes           efInsertPushes
Map control chars       efMapCtrls
Pad character           ---------
Parens for minus        efParensForMinus
Required                efRequired
Right justify           efRightJustify
Suppress zeroes         sefSuppressZero
Trim blanks             efTrimBlanks
```

Delimiters
Set left and right field delimiters. If either delimiter is #0, neither delimiter will be displayed and the menu will show 'None'.

Mouse support
Enable mouse support in program being designed (does not affect MAKESCRN except in Test mode).

Password char
Set character used to represent text in password mode.

Scroll by page
Activate esMousePage option (Mouse), esScrollByPage option (Kbd), Both, or neither (No).

Wrap mode
Set the wrap mode for the entry screen (WrapAtEdges, StopAtEdges, ExitAtEdges).

Fields

Add
Add a new entry field or text field at the cursor position (see "Adding and Modifying Fields" and "Adding and Modifying Text Fields," below).

Copy
Copy the field (entry or text) at the cursor to a new location. This command is useful when you need to define several fields that are largely identical (same picture masks and options).

Edit
Edit the definition for the field at the cursor (see "Adding and Modifying Fields" and "Adding and Modifying Text Fields," below).

Move
Move the field (entry or text) at the cursor to a new location.

Remove
Remove the field (entry or text) at the cursor.

Object

Load
Load a new or existing existing entry screen from the current library.

Save **<F2>**
Save the current entry screen.

Rename
Change the name of the current entry screen.

Save as
Save the current entry screen under a new name.

Info **<AltI>**
Display information about the object being created. Note that, if no object is loaded, <AltI> will display information about the current library instead (see Library-Info, below).

Test **<AltT>**
Test the entry screen being designed.

Write source
Generate source code for the current entry screen.

Document
Generate documentation for the current entry screen (see "Generating Documentation").

Library

Load **<F3>**
Load a new or existing library. If the file specified does not exist, you will be asked if you want to create it.

New
Create a new library. If the file specified already exists, you will be asked if you want to overwrite it.

Delete
Displays a list of all objects in the library. Press <Enter> to mark an object for deletion (a checkmark will appear beside the name) or to remove an existing deletion mark. Press <Esc> when you are finished marking object names. Note that the objects will not be physically removed from the library until it is packed.

Info **<AltI>**
Display information about the current library. Note that <AltI> will execute this command only if no object is loaded; otherwise, it will display information about the current object (see Object-Info).

Pack
Pack the current library if packing is needed. Packing purges the library of earlier versions of objects, as well as objects marked for deletion.

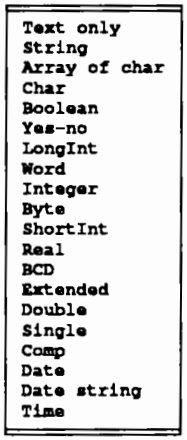
Quit **<AltX>**
Quit the program.

Adding and Modifying Fields

Although we've already shown you the basic steps involved in adding a field to an entry screen, we skipped a lot of the details. This time we'll go through it step by step.

Step 1 is to move the cursor to the location on the screen where you want the first character of the prompt to appear. (You must be in Edit mode.)

Step 2 is to press <F10> to enter the menu system, then select Fields-Add. When you do, the following submenu will appear:



Ignore for now the first choice, 'Text', which we'll discuss a bit later--it's for adding text fields. The remaining choices pertain to entry fields. Move the cursor to the name of the field type that you want to add and press <Enter>.

Step 3 is the hard part. The field definition screen that we saw earlier will now appear. We'll go over each of the fields in this entry screen individually. Note that not all of them will appear in every situation: a string field, for example, does not have lower and upper "limits." The fields are grouped below by sections, corresponding to the sections of the entry screen itself.

Name

Name
Specifies the symbolic name to be used to refer to this field. The name must be a valid Pascal identifier (MAKESCRN checks this), and it must be unique--no two fields should have the same name (MAKESCRN does *not* check for this). This field is optional. If you do not specify a name, the source code generated by MAKESCRN will refer to the field as 'FieldXxx' where 'Xxx' is the field's ID number (e.g., 'Field001').

Prompt

Prompt
Specifies the prompt for the field, if any. If the coordinates for the prompt and the field are the same at the time the prompt is entered, the column coordinate for the field will be updated automatically to position the field after the prompt (with one space between them). The same thing will happen if a susequent change to the prompt causes it to overlap the field.

Row, Col
The coordinates for the prompt.

Field

Type
Specifies the type of the field (String, Byte, etc.). Normally this field is protected, but it is actually a multiple choice field, and in a few cases you are allowed to change the field to a similar type. These are the types that are considered "similar":

- Boolean and YesNo
- LongInt, Word, Integer, Byte, and ShortInt
- Extended, Double, Single, Comp

Editor
Specifies the field editor to be used. In some cases (such as Date), there is only one choice available here, so the field is protected. But in others there are up to three: "Normal" (StringEditor or CharEditor), "Simple" (SimpleStringEditor or SimpleCharEditor), or "Numeric" (NumberEditor). This too is a multiple choice field.

Picture
Specifies the picture mask for the field. See "Defining a Picture Mask," below.

Row, Col
The coordinates for the field itself.

Width
Specifies the width for the field, if it is different from the length of the picture. This value, which applies only to string and array of char fields, is filled in for you automatically when you enter a picture mask. You should change it only if you want the field to be able to scroll horizontally. The width may never be greater than the length of the picture mask or the width of the window. Press <F10> or <ClickLeft> to reset the width to the current length of the picture mask.

Dec places
Specifies the maximum number of decimal places to display for a real field (i.e., Real, BCD, Extended, Single, Double, or Comp). This value is ignored if the picture mask has a fixed decimal place.

Pad char
Specifies the character used to pad the end of the field when it is displayed or edited. This setting does not affect the actual contents of the field, only the way it is displayed. If you want to choose the character from a list, press <F10> or <ClickLeft>.

Options
This is a nested field that leads to a menu of field options that is largely identical to the Fields-Options submenu. Press <F10> or <ClickLeft> to invoke the menu.

Range

Lower limit
The lower limit for the field. This field and the next one are hidden if the field being added does not have a range.

Upper limit
The upper limit for the field. May not be less than the lower limit. If the two are equal, no range limits are imposed on the field value.

Attributes

The last section of the entry screen displays the seven pairs of video attributes that affect the way the field is displayed. (A "pair" consists of two video attributes, one for color systems and one for monochrome.) The first three pairs pertain to the prompt and specify the attributes to be be used when it is unselected (Normal), Selected, or Protected. The second three pairs give the same information for the field itself. The last pair specify the attributes to use for control characters when the efMapCtrls option is on. You can change a given attribute either by entering the appropriate value in hex (e.g., '0F' for White on Black) or by pressing <F10> or <ClickLeft> and selecting it interactively.

To edit an existing field, position the cursor somewhere on the desired field, then press <Enter>. The field definition screen will appear, and you can modify it however you wish.

To move an entry field, position the cursor somewhere on the desired field, then press <F10> and select Fields-Move. A box cursor will appear at the start of the prompt for the field. Move the cursor to the new location for the start of the prompt and press <Enter>. MAKESCRN will report an error if you try to move the field on top of another one.

Defining a Picture Mask

The most critical part of the field definition is the picture mask, and there are lots of ways to mess it up. It would literally take a program-within-a-program to catch all the possible errors that can be made when defining a picture mask, and to some extent you are responsible for entering one that is valid and appropriate. MAKESCRN does, however, do four things to try to help you:

1. It supplies a reasonable default value to get you started.
2. It has a special validation routine for this field that is designed to catch the most blatant errors. If it finds one, it will either report the error to you or, if the required change is obvious, make it for you.
3. It provides you with a list of sample picture masks to remind you of what the various mask characters are and what they mean. When the cursor is on the Picture field, you can see this list by pressing <AltF10>. If you want MAKESCRN to plug one of the sample picture masks into the field for you, move the highlight bar to the appropriate item and press <AltI> to Insert it.
4. It provides the <F10> shortcut we used earlier in the tutorial. If the picture mask will consist of a single mask character, you can press <F10>, then enter the mask character, followed by the length of the picture mask.

When defining a simple field (one that uses SimpleStringEditor or SimpleCharEditor), enter the picture mask just as you would if it were not a simple field. For example, if the field is of type Word and it needs to be 5 characters wide, enter '99999' as your picture mask. Only the first character in the picture mask is actually meaningful, but its length is used to determine the width of the field. MAKESCRN will report an error if the picture mask for a simple field does not consist of a single character.

Adding and Modifying Text Fields

Adding a text field to an entry screen is quite a bit simpler than adding an entry field. There are only three steps involved:

1. While in Edit mode, move the cursor to where you want the text field to start, press <F10>, then select Fields-Add-Text.
2. Enter the text.

3. Select the attributes to be used on color and monochrome systems.

To edit an existing text field, move the cursor until it is on some part of the text field, press <Enter>, then repeat steps 2 and 3.

You move a text field in exactly the same way that you move an entry field.

Edit Mode Commands

The following commands are available while in Edit mode:

<Left>, <CtrlS>
Cursor left one column.

<Right>, <CtrlD>
Cursor right one column.

<CtrlLeft>, <CtrlA>
Cursor left one word.

<CtrlRight>, <CtrlF>
Cursor right one word.

<Up>, <CtrlE>, <CtrlW>
Cursor up one row.

<Down>, <CtrlZ>, <CtrlX>
Cursor down one row.

<Home>, <CtrlQ><S>
Cursor to column 1.

<End>, <CtrlQ><D>
Cursor to end of line (just beyond end of last field on the line).

<PgUp>, <CtrlR>
Cursor up one page.

<PgDn>, <CtrlC>
Cursor down one page.

<CtrlPgUp>, <CtrlQ><R>
Cursor to row 1, column 1.

<CtrlPgDn>, <CtrlQ><C>
Cursor to end of last line with a field on it.

<CtrlHome>, <CtrlQ><E>
Cursor to top row of window.

<CtrlEnd>, <CtrlQ><X>
Cursor to bottom row of window.

<Tab>
Cursor to next tab stop (every 8 characters).

<ShiftTab>
Cursor to previous tab stop (every 8 characters).

<Enter>, <ClickLeft>
Edit the field at the cursor.

<CtrlN>
Insert line at cursor. Moves all fields on current row and beyond down one row.

<CtrlY>
Delete line at cursor. Deletes all fields on current row (after asking for confirmation), and moves all fields on subsequent rows up one row.

<Esc>, <ClickRight>
Exit from Edit mode; invokes main menu.

Moving and Resizing Windows

The following commands are available while moving and resizing a window. Note that, if a mouse is installed, the cursor can also be moved with the mouse.

<Left>, <CtrlS>
Cursor left one column.

<Right>, <CtrlD>
Cursor right one column.

<Up>, <CtrlE>, <CtrlW>
Cursor up one row.

<Down>, <CtrlX>, <CtrlZ>
Cursor down one row.

<Home>, <CtrlQ><S>
Cursor to column 1.

<End>, <CtrlQ><D>
Cursor to last column.

<ShiftTab>, <CtrlLeft>, <CtrlA>
Cursor to previous horizontal tab stop (every 8 columns).

<Tab>, <CtrlRight>, <CtrlF>
Cursor to next horizontal tab stop (every 8 columns).

<PgUp>, <CtrlR>
Cursor to previous vertical tab stop (every 5 rows).

<PgDn>, <CtrlC>
Cursor to next vertical tab stop (every 5 rows).

<CtrlHome>, <CtrlPgUp>, <CtrlQ><E>, <CtrlQ><R>
Cursor to top row.

<CtrlEnd>, <CtrlPgDn>, <CtrlQ><C>, <CtrlQ><X>
Cursor to bottom row.

<AltH>
Center window horizontally.

<AltV>
Center window vertically.

<AltC>
Center window both horizontally and vertically.

<Enter>, <ClickLeft>
Accept size/position.

<Esc>, <ClickRight>
Cancel changes.

<ScrollLock>
Toggle between move and resize modes (Resize only).

Generating Source Code

As we've seen, there's not much involved in generating source code. You simply select the appropriate menu choice and enter the name of the source file to be generated. If the file specified already exists, you will be asked if it should be overwritten. Be careful here. You don't want to overwrite a "real" program of the same name. For this reason, it is generally a wise practice to give your entry screens a different name than the program it will ultimately be used in, especially if all your files are in the same directory.

We should reiterate that there are four menu choices that affect the source code that is produced by MAKESCRN: "Color constants," "Help constants," "User proc hooks," (Defaults) and "Mouse support" (Options). These choices, explained earlier, all correspond to options that are On by default.

Generating Documentation

In addition to generating source code, MAKESCRN can also generate "documentation": a brief report showing information about the entry screen and the fields that are in it. If you were to generate documentation for the sample entry screen ENTRY1 (in SAMPLE.OPL), the first part of the report would look something like this:

```
Entry Screen Documentation
--------------------------
  Object:                 ENTRY1
  Library:                SAMPLE.OPL
  Size when last saved:   1081

  Text fields:            0
  Entry fields:           5
  Scrollable:             No

Field name           Type / Editor Prompt / Picture                    Row Col
-------------------- ------------- ---------------------------------- --- ---
 Name                String        'Name:'                             2   5
                     Normal        'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'  2  11
```

The first section of the report shows information about the entry screen as an object: its name, the library it's in, and its size.

The second section shows the number of text fields in the entry screen, the number of entry fields, and whether or not it is scrollable (i.e., is it an EntryScreen or a ScrollingEntryScreen).

The last section is a list of the fields in the entry screen. If there are any text fields, this section is actually divided into two parts: one for text fields and one for entry fields. For each text field, the documentation shows its ID number, its coordinates, and its contents. For each entry field, the following information is given: its symbolic name (or 'FieldXxx' if there isn't one); its type

(String, Byte, etc.), its prompt, the coordinates for the prompt, the field editor used (Normal, Simple, Numeric), its picture mask, and the coordinates for the field itself.

Normally, the report is sent to a text file with an extension of "DOC". If you want it sent to the printer instead, enter "PRN." (or "LPT1." or "LPT2.") when prompted for a filename.

Limitations

Probably the most noteworthy "limitation" of MAKESCRN is that it is just plain big. The pre-compiled version included on the distribution disk requires approximately 400K of memory *minimum*. If this prevents you from running it, you might consider recompiling it after setting the conditional compilation directives in OPDEFINE.INC as follows:

```
{$N-}
{.$DEFINE UseMouse}
{.$DEFINE UseBCD}
{.$DEFINE UseDates}
{.$DEFINE PickListFields}
```

This will disable some of the features of MAKESCRN, but it will definitely make it smaller. Be forewarned, however, that you will need even more memory to compile it than you do to run it--roughly 450-500K. If you are able to recompile the program with these settings, however, the resulting EXE file will be about 60K smaller and will use roughly 55-60K less memory.

If you do decide to recompile MAKESCRN, be certain that these other directives *are* defined as follows:

```
{.$DEFINE UseCrt}
{$DEFINE UseScrollBars}
{$DEFINE UseHotSpots}
{$DEFINE UseShadows}
{$DEFINE UseAdjustableWindows}
{$DEFINE FourByteDates}
```

Enabling the UseCrt directive would do nothing except make the program larger. The next four directives are assumed to be enabled. If they are not, MAKESCRN will not compile. We strongly discourage you from changing FourByteDates. There is nothing to be gained by doing so, and disabling it may cause problems later if you try to switch to a version of MAKESCRN that was compiled with it enabled.

MAKESCRN is also limited in the sense that it does not support all the features of OPENTRY. Some of the less commonly used field options cannot be selected, for example, nor can it generate working pre-edit, post-edit, global validation, or help routines. More importantly, it does not support the following field types:

- multi-line fields
- pick fields
- multiple-choice fields
- nested fields
- user-defined fields

2. ENTRY

Although it doesn't do anything useful, the demonstration program ENTRY.PAS is noteworthy because it offers examples of how to use several of the units in **Object Professional**--notably OPDATE, OPENTRY, OPPICK, OPHELP, and OPMEMO--as well as some of the basic programming hooks made available by all the command processing objects. It demonstrates, for example, how to create a sophisticated data entry screen with comprehensive input validation; how to implement a context-sensitive help system; how to use a pick list within an entry screen; how to add a memo field to an entry screen; how to perform background tasks while waiting for keyboard input; how to add and process user-defined commands; and how to write programs that adapt to different currencies and date/time formats.

To run the program, you need two files: ENTRY.EXE and ENTRY.HLP (which contains the data used by the help system). When you load the program, you'll see that the screen is divided into three sections. The top section is a box divided into two parts, the second of which serves as a status line, showing the current record number, the date, and the time. The date and time are kept current by a user-defined keyboard input routine that displays them as a background task. The middle section is a data entry screen consisting of several fields, which we'll discuss in a moment. The bottom section is also a box divided into two parts. The first part is a help line that provides brief instructions on what should be entered in the current field; the help text changes as you move from field to field. The second part is a static help line that tells you what keys perform some of the basic functions.

The main entry screen has fifteen fields. The list below draws attention to any special qualities of a given field.

Name: The one thing truly special about this field is that the first character in each word that you enter is automatically converted to upper case. The field's only other noteworthy behavior applies to most of the other fields in the entry screen: If the field already contains text when you move to it, and the first key pressed is an ASCII character, the field will be cleared. If the field editor is in insert mode, text at the very end of the field can be pushed off the end. Press <Enter> or <Tab> to move to the next field when you're finished.

Address: This field's only special characteristic is that it is marked as required, meaning that it cannot be left blank.

City: Unlike other fields in the entry screen, text at the end of this field cannot be pushed off the end while in insert mode.

State: This field, which was added to the entry screen with AddPickStringField, is "edited" using a pick list created with OPPICK. Rather than entering an abbreviation for a state name, you can simply select one from a list of valid abbreviations.

Zip: You can enter either a five- or a nine-digit zip code here. Only numbers and spaces may be entered in this field, and an error message will be displayed if you try to enter a zip code with something other than 5 or 9 digits.

Phones: This is a dummy field that leads to a second (nested) data entry screen. To get to the second screen, press <Enter> while the cursor is on this field. The two fields in the entry screen are discussed below.

Gender: This is a multiple choice field. To select from the various choices (Unknown, Male, Female), press <Space> or <+> to go forward through the list of choices. The <-> key takes you through the list backwards.

Married: This is a yes/no field. Press <Y> or <N>, or <Enter> to accept the default answer.

Born: This is a date field. You can enter any date between 1/1/00 (1900) and 12/31/99 (1999) here. The validation routine for the field insures that the date entered is valid, and you'll get an error message if you try to enter an invalid one (like 1/32/88).

Age: This is a calculated field, so you can't change it. Using one of the hooks provided by OPENTRY, the value for this field is calculated and displayed whenever you update the Born field. This field is also right-justified.

Hourly wage: This is a numeric field, meaning that numbers are entered from right to left in calculator fashion. The decimal point is fixed, and the dollar sign floats.

Weekly: Like Age, this is a calculated field. Its value is equal to "Hourly wage" * "Hours/week". The decimal point is fixed, and the dollar sign floats. The field is padded with asterisks rather than spaces.

Hours/week: This is a special kind of multiple choice field. Rather than toggling though a list of choices, you increment or decrement the default value using the <+> and <-> keys. As the value changes, the values of the Weekly and Yearly fields are also updated to reflect the change.

Yearly: This too is a calculated field. Its value is equal to "Weekly" * 52. The decimal point is fixed, and the dollar sign floats. The field is padded with asterisks rather than spaces.

Notes: This is a memo field created with OPMEMO. If you press <Enter> when the cursor is on this field, a window will pop up over the entry screen that allows you to enter up to 2K of text.

The second entry screen, accessed by moving the cursor to the Phones field and pressing <Enter>, has two fields.

Work phone: Only numbers and spaces may be entered in this field. A valid entry will consist of a complete (seven-digit) phone number, with or without an area code. If you try to enter a one or two-digit area code, or an incomplete phone number, an error message will be displayed. The field is padded with underscores ('_') rather than spaces during editing.

Home phone: This field has the same characteristics as the Work phone field.

ENTRY is also notable for its context-sensitive help system, developed with OPHELP. As you move the cursor from one field to another, try pressing the <F1> key for details on that particular field. Words or phrases highlighted in purple (on color screens) or reverse video (on mono screens) are links to related help topics. Pressing <Enter> will display the help screen for that topic. You can then return to the previous topic by pressing <AltF1>. You'll notice that some help screens have more than one link: if you see a word or phrase displayed in yellow on blue (on color screens) or one that is underlined (on mono screens), you can select that topic by moving the cursor to that word and pressing <Enter>. The help screen for the Hourly wage field, for example, has four links.

To see an index of all the help topics, press <F1> from anywhere in the help system. You can then choose any of the topics by moving the highlight bar to the appropriate topic and pressing <Enter>. A few of these topics (those at the beginning of the list) cannot be accessed except from the index. Among these are topics concerning the various editing and cursor movement commands that are available.

Although the help system briefly describes each of the available commands, there are a few worth drawing your attention to here:

<AltB>
Normally when you enter an illegal character, ENTRY will beep at you. To keep it from doing so, press <AltB>, which toggles the error bell on or off. This is another example of a user-defined command.

<Tab>, <ShiftTab>
These keys move the cursor to the next or previous subfield, which is not necessarily the same thing as the next field. The Born field, for example, has three subfields: month, day, and year.

<Up>, <Down>
These keys move the cursor up or down on the screen, not to the next or previous field.

<PgUp>, <PgDn>
Pressing these keys move you to the next or previous record. Although you can enter data for up to 10 records, keep in mind that ENTRY does not provide any way to save these records. Don't waste too much time filling in the various records.

<Esc>, <CtrlBreak>
This restores the previous contents of the current record and gives you a chance to abort the editing session.

<CtrlEnter>, <CtrlK><D>, <CtrlK><Q>
Pressing any of these keys ends the data entry session. Although you can't tell it, the previous contents of the current record are not restored as they are when you press <Esc>.

Thanks to the low-level facilities furnished by OPMOUSE, and the higher-level ones in several other units, ENTRY is also able to support the mouse in fine fashion. While editing fields, you can do the following things using a mouse (if one is installed):

- Move the mouse cursor (shaped like a diamond) to any unprotected field on the data entry screen and select it by clicking the left mouse button
- Move the mouse cursor to the Phones field and click the left mouse button: the secondary entry screen associated with it will be popped up
- Move the mouse cursor to the State field and click the left mouse button to pop up the pick list of state names
- Move the mouse cursor to the Marital status field and click the left mouse button once to select it, a second time to toggle the value
- Move the mouse cursor to the Age or Hours/week fields and click the left mouse button once to select the field, a second time to increment the value
- Press the right mouse button to simulate pressing <Esc>
- Press both mouse buttons (the left and the right) to bring up the help screen for the current field
- Press the center mouse button (on three-button mice only) to simulate pressing <Enter>

You can also use the mouse when selecting from pick lists (see the section on OPPICK for details), when using the help system (see the section on OPHELP), and when responding to prompts for a single keypress (in general, clicking the right button does the same thing as pressing <Esc>, and clicking the left button does the same thing as pressing <Enter>).

3. BEST

BEST stands for Big Entry Screen Test, and that's really what this program is: a test program that we developed primarily for our own use to test the ScrollingEntryScreen object and some of the other features of OPENTRY. Nevertheless, it does demonstrate several useful techniques. And since it is the only example of a scrollable entry screen beyond those in the manual, you will definitely want to look at it if you are going to work with the ScrollingEntryScreen.

The first thing you should do when you run BEST is to change the value of the first field to a smaller number, say 12. This value indicates the number of rows of fields that are visible and accessible to the user. The default value of 68 allows access to all the fields in the entry screen. If you try to move the cursor beyond the row at the bottom, the entry screen will scroll to display additional fields. If you've changed the value of the first field to 12, however, you won't be able to move the cursor beyond the 13th row (counting the first), nor will you be able to see any of the fields on rows 13-69. This trickery is performed by a post-edit routine that dynamically hides and unhides fields.

Change the value of the first field back to 68, move the cursor to the row starting with 'Field 103', press <Down>, and then <Up>. When you press <Down>, a whole new set of fields will appear in the window. Now press <AltS> and do the same thing. This time the window will scroll by only one line, rather than by a full "page." This demonstrates what happens when the esScrollbyPage option is on and off, respectively.

Now you can try experimenting with some window-related commands. <AltZ> "zooms" and "unzooms" the entry screen's window. When it is zoomed, the entry screen occupies all but the last row of the screen. If the window is not zoomed, <AltM> lets you move the window around the screen by pressing the cursor keys, and <AltR> lets you resize it.

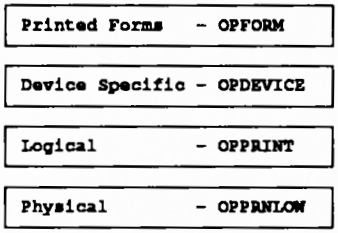
If you have a mouse, you can do several things with it here. You can select a particular field by moving the mouse cursor to it and clicking the left button. You can scroll the window by clicking left on the scroll bar. You can zoom or unzoom the window by clicking on the up arrow displayed near the top right hand corner of the window border. You can move the window by clicking on the window's title, centered at the top of the border: move the mouse until the window is where you want it, then click left again. You can resize the window by clicking left on the bottom right corner of the window, just below the scroll bar: again, move the mouse until the window looks the way you want it and then click left. And, finally, you can close the window by clicking left on the hot spot near the top left hand corner of the window (clicking the right button would do the same thing).

7. Printing

A. Overview

Printer support is one of the trickiest issues a programmer must deal with when writing applications. Few printers are compatible with each other, and some are much more sophisticated than the rest. In addition to communicating with the printer, an application has to format the output for an organized and pleasing appearance. **Object Professional** provides a powerful set of units designed to tame these printer issues.

Four units are used to implement the printer and printed forms support provided by **Object Professional**. The following diagram helps to illustrate the purpose of these units:



The OPPRNLOW unit provides the physical foundation level. It defines how character output is written and how printer status information is retrieved and interpreted. Although you might expect that these low-level functions would be assembly language macros or even traditional procedures, OPPRNLOW is a collection of objects and it gains important advantages from OOP.

OPPRNLOW's objects start with an abstract one called BasePrinter, which defines the fundamental operations for printer output–initialization, opening for output, character translation, writing a character, flushing, and closing the device. Objects derived from BasePrinter implement these operations in different ways: DosPrinter uses DOS file services for output, and BiosPrinter uses the BIOS. The pros and cons of these two objects are discussed later.

One of the most important benefits from the OOP approach arises in printer error handling, which is probably the trickiest issue of all. How do you determine whether the printer is ready to accept output? Unfortunately there is no 100% effective test. OPPRNLOW provides the best answers you'll find anywhere by relying on virtual methods and procedure pointers to provide plenty of flexibility.

Another important feature of OPPRNLOW is its ability to connect the BasePrinter object (and its derivatives) to a standard Text file device driver. In this way the full formatting power of the Write and WriteLn statements can be used to write printer output with high performance and tight error control.

OPPRINT provides the next level of printer support: the ability to print in different fonts, at different spacings, in different modes. Printers usually understand a small programming language of their own which has a cryptic grammar of control codes and escape sequences. Some printers have a very small set of commands, others an entire page definition language. Taking advantage of these varying capabilities without unduely constraining an application is a tough programming challenge.

OPPRINT again takes an object-oriented approach in order to provide the required degree of flexibility. OPPRINT's primary object is the Printer type. Printer includes a pointer to a BasePrinter object and thus inherits the physical communication capabilities from OPPRNLOW. More importantly, Printer manages a linked list of PrintMode objects. Each PrintMode knows how to enable and disable a specific printer characteristic, and can identify itself by name, by identification number, or by attributes such as width and height of a font. The Printer object also contains special hooks for printer characteristics that require more than control codes, like line-drawing and shading.

OPDEVICE provides the device-specific information that OPPRINT needs to support particular printers. OPDEVICE includes the control codes, character translation tables, and custom hook routines for common printers like Epson, Diablo, and LaserJet. Specific capabilities include support for bold, italic, compressed, and underline type; line spacing and page size control; and line drawing when the printer can support it. OPDEVICE doesn't support anywhere close to the number of printers that WordPerfect does, but we expect to enhance it as time goes by.

OPFORM provides the highest level of printer support. OPFORM is to the printed page what OPENTRY is to the video screen. With OPFORM you format print fields by combining raw data with a printer mode, then position the fields on a page, sequence the pages into a form, and merge the forms into a report.

Like OPENTRY, OPFORM provides a broad assortment of objects for converting Pascal data types–strings, words, reals, booleans, etc.–into print fields. Each print field has a position, a print mode, a picture mask, and the capability to print itself. The PrintBlock object manages a list of print fields. PrintPage combines a list of PrintBlocks with page control mechanisms, headers, and footers. A PrintedForm object manages a list of PrintPages. It also contains a pointer to a Printer object (where it obtains device-specific knowledge), information about margins and options, and a number of special hooks for additional control. Finally, a Report object contains a list of PrintedForms.

As you can tell, this chapter has plenty to cover. You may not need all of it. Many applications will be adequately served by the flexible output and error detection capabilities of OPPRNLOW. Others will find that OPPRINT's mode control and printhead positioning are enough to get the job done. If you need comprehensive report generation facilities, read on through the OPFORM section. The chapter ends with a description of FORMTEST, a sample program that combines the capabilities of all four units.

B. OPPRNLOW: Physical Level Printer Support

Turbo Pascal provides the standard PRINTER unit for sending output to a printer, but this unit provides access only to the DOS PRN device and has very limited error handling capabilities. OPPRNLOW is designed to provide much more flexibility, including support for all LPT devices (LPT1, LPT2, LPT3, as well as PRN and disk files), flexible error handling, automatic character translation, and more.

OPPRNLOW implements several objects for low-level printing support. It also provides a way to associate these objects with a Turbo Pascal Text File Device Driver (TFDD). Using these facilities, an application program can improve its printer support without having to give up the convenience of Write and WriteLn.

The following hierarchy shows the objects provided by OPPRNLOW:

```
BasePrinter
  DosPrinter
  FlexiblePrinter*
    BiosPrinter
```

Although OPPRNLOW is a low level unit that the other printer units build upon, the DosPrinter and BiosPrinter objects, and especially the associated TFDD support, are useful in their own right.

The BasePrinter Object

The BasePrinter object is an abstract object type used to derive other printer objects. It defines the methods and data fields that will be common to derived printer objects. OPPRINT and OPFORM declare all printers as pointers to BasePrinter, allowing any object type derived from BasePrinter to be used.

The BasePrinter object has data fields that contain information about the type of printer, a "handle" to the printer (which may be a DOS file handle, an LPT number, or some other word sized or smaller value), information regarding the last error code and operation, the number of characters written by the last output call, and a character translation table.

Many of the methods defined in the BasePrinter object are virtual so that they can be overridden by derived types. The virtual methods include:

- open and close the device
- translate characters
- output single characters and blocks of characters
- translate raw errors to logical error codes

Note that BasePrinter is not capable of printing by itself. It just sets the stage for the following objects. We'll document only a few of BasePrinter's virtual methods: those that an application will call directly and those that you may have need to override.

Since many of the methods of DosPrinter and BiosPrinter produce the same results (although they are implemented in different ways), we'll document them a single time only in the section for BasePrinter. Only those methods unique to DosPrinter or BiosPrinter will appear in their method pages.

The DosPrinter Object

The DosPrinter object uses standard DOS function calls to send characters to a DOS file or character device. DOS character devices include ones like PRN, LPT1, CON and AUX.

There are pros and cons to using DOS services for printer output. Advantages include:

- switching output between a file and device is transparent to the application
- it's cleaner to use DOS rather than a lower level like the BIOS
- tends to work better with unsophisticated print spoolers and older networks

Among the disadvantages are:

- printer status checking is limited to the hard-coded DOS IOCTRL call
- speed is significantly lower than BIOS-level printing

We believe that the disadvantages outweigh the benefits in the case of **Object Professional,** since OPPRNLOW allows transparent switching between file output and printer output anyway, and since the PC world has clearly standardized at BIOS-level compatibility.

Here's how to use the DosPrinter object:

```
program DosPrtExample;  {EXPRNLW1.PAS}
uses
  OpRoot, OpPrnLow;
const
  Hello : String[12] = 'Hello World'^L;
var
  MyPrinter : DosPrinter;
begin
  if not MyPrinter.Init('PRN', Prn) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus);
    Halt;
  end;
  MyPrinter.PrnPutBlock(Hello[1], Length(Hello));
  MyPrinter.Done;
end.
```

This example initializes a DosPrinter to write to the DOS standard printer and sends a block of characters to it. The block of characters sent is the text "Hello World" followed by the form feed character (^L).

The FlexiblePrinter Object

This is a printer type that probably will not be used directly. It uses procedure and function pointers to provide a flexible and adaptable means of outputting characters. The reason for using pointers is that the object's behavior can be changed *after* the object has been instantiated. We've taken advantage of this behavior in BiosPrinter, which is derived from FlexiblePrinter. We won't document FlexiblePrinter in any more detail; if you think you need its capabilities, take a look at OPPRNLOW.PAS.

The BiosPrinter Object

BiosPrinter uses BIOS interrupt $17 to communicate with the printer. The BIOS provides three functions for interrupt $17: function 0 to send characters to the printer, function 1 to reset it, and function 2 to return status information. The BIOS also supports three LPT devices: LPT1, LPT2 and LPT3. You can specify any of these when you initialize a BiosPrinter object, and you can even switch among them later by calling the SetLptPort method.

One of the advantages of using the BIOS for printing is that direct control over error handling becomes possible. The basic strategy is to call interrupt $17 function 2 to return a status byte just before printing each character. If the status byte indicates that the printer is ready then call

function 0 to write the character, otherwise generate an error. In this way errors are dealt with immediately; correctable errors can be fixed and printing resumed with no loss of output.

There is a catch, however. The status byte returned by function 2 is usually a direct bit-mapped image of the status lines of the computer's parallel port. (When the MODE command is used to redirect printing to a serial port, MODE's serial driver returns a byte value that simulates the results of the parallel port.) Even though the meaning of each bit has nominally been standardized, the printer manufacturers of the world (and the authors of the MODE command) have not agreed upon a standard set of byte values to indicate each possible printer condition. Worse yet, when a parallel port is not connected to a printer at all, the status lines are floating and can return arbitrary values.

As a result, interpretation of the function 2 status byte is an art. To allow for variations, BiosPrinter lets you select among several alternate tests. In fact, by overriding BasePrinter's PrnXlatErrorCode method, you can substitute your own routine for interpreting printer errors, but that should rarely be necessary.

The raw material we have to work with is the status byte, whose bits are defined as:

Bit	Mask	Meaning when set	Name for mask
0	$01	timed out	bpTimedOut
1	$02	--	
2	$04	--	
3	$08	I/O error	bpIOError
4	$10	printer selected	bpSelected
5	$20	out of paper	bpOutOfPaper
6	$40	acknowledge	bpAcknowledge
7	$80	not busy	bpNotBusy

In an ideal world, we would want to assure that bits 0, 3, and 5 were clear and that bits 4 and 7 were set. (Bit 6, acknowledge, is set only briefly and is not used for status checking.) This test is too stringent, however; almost no printers pass it and therefore no programs use it.

BiosPrinter provides five different ways to test these bits. The test to use is specified by passing a test number to the object's InitCustom constructor.

Test Description

0 no test at all. The printer is always considered to be ready.

1 considered to be the best test based on recent empirical surveys. It confirms correct values for the timed out, I/O error, out of paper, and not busy bits as follows:

```
ErrorCode := ErrorCode xor $80;
PrinterReady := ((ErrorCode and $A9) = 0);
```

This test is used by the print screen interrupt handlers of many recent BIOS versions.

2 slightly looser than test 1 in that it ignores the not busy bit:

```
PrinterReady := ((ErrorCode and $29) = 0);
```

3 a relaxed version of test 1 that ignores the timed out and I/O error bits:

```
ErrorCode := ErrorCode xor $80;
PrinterReady := ((ErrorCode and $A0) = 0);
```

4 a user-customizable test that AND's the status byte with a mask. If the result equals the mask, the printer is considered to be ready:

```
PrinterReady := ((ErrorCode and Mask) = Mask);
```

Many programs use this test with a mask value of $10 (to test the selected bit only) or $90 (to test selected and not busy).

Here's how to use the BiosPrinter object:

```
program BiosPrtExample;  {EXPRNLW2.PAS}
uses
  OpRoot, OpPrnLow;
const
  Hello : String[12] = 'Hello World'^L;
var
  MyPrinter : BiosPrinter;
begin
  if not MyPrinter.Init(Lpt1) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus);
    Halt;
  end;
  MyPrinter.PrnPutBlock(Hello[1], Length(Hello));
  MyPrinter.Done;
end.
```

This example initializes a BiosPrinter to write to LPT1 using the default printer ready check (test 1). An alternate constructor (InitCustom) allows you to select your favorite test.

You may find that none of the tests provided by BiosPrinter work in all situations. If your application has broad distribution and must work with an unlimited number of printers, you might consider the "calibration" approach to error detection. With this approach you supply a calibration program which asks the user to put the printer into various different states (offline, out of paper, disconnected, powered off, and ready) and then stores and evaluates the status bytes for each state. We've provided an example of a calibration program on the BONUS diskette. Decompress CALIBRAT.LZH for more information.

Don't get your hopes too high, even with calibration. The most severe problem we've run into is that the MODE command supplied with DOS versions prior to 3.2 often returns exactly the same status value whether the serial printer is online, offline, or totally disconnected. If you run into this situation, the best approach is to recommend that your users upgrade to a newer version of DOS.

Actually, an even lower level approach to printing is possible: direct port access. While improved error checking is available at this level, the disadvantages are severe. Direct port access is substantially different for parallel and serial ports, and it is not compatible with print spooling, networks, or multitasking environments like DesqView. None of **Object Professional**'s tools work at this level.

Customized Printer Objects

This chapter has focused on flexibility as a key factor in robust output device support. In this section we'll derive an object type that illustrates some of the benefits of object-oriented printer support. This example assumes that your application is designed to run on a network and that you need to support both a network printer and a local printer for each workstation. The network is assumed to be compatible with Novell's NetWare print spooling functions.

```
type
  NetworkPrn =
    object(BiosPrinter)
      Spooling : Boolean;
      constructor Init(LPTNo : LPTType; SpoolingOn : Boolean);
        {-Initialize a printer}
      destructor Done; virtual;
        {-Done with a printer}
      procedure SpoolOn;
        {-Enabled spooler}
```

```
      procedure SpoolOff;
        {-Disable and flush spooler}
    end;

constructor NetworkPrn.Init(LPTNo : LPTType; SpoolingOn : Boolean);
begin
  if not BiosPrinter.Init(LPTNo) then
    Fail;
  Spooling := SpoolingOn;
  if SpoolingOn then
    SpoolOn;
end;

destructor NetworkPrn.Done;
begin
  if Spooling then
    SpoolOff;
  BiosPrinter.Done;
end;

procedure NetworkPrn.SpoolOn;
var
  Regs : Registers;
begin
  {set default local printer}
  with Regs do begin
    AX := $B805;
    DH := Byte(GetLptPort);
  end;
  MsDos(Regs);    {NetWare set local LPT function}
  {turn on print spooler}
  with Regs do begin
    AX := $DF00;
    DX := $00;
  end;
  MsDos(Regs);    {NetWare start LPT capture function}
end;

procedure NetworkPrn.SpoolOff;
var
  Regs : Registers;
begin
  {turn off print spooler}
  with Regs do begin
    AX := $DF00;
    DX := $01;
  end;
  MsDos(Regs);    {NetWare end LPT capture function}
end;
```

NetworkPrn adds new functionality to the BiosPrinter type. Network spooling can be enabled when the object is initialized or toggled at any time.

The NetworkPrn example provides an example of adding to one of the existing objects. It may also be desirable to override one of the virtual methods to obtain new behavior. The most likely method to override is PrnXlatErrorCode, which translates a raw error code into a logical printer status. Here's an example:

```
type
  SpecialBiosPrinter =
    object(BiosPrinter)
      function PrnXlatErrorCode(Call : PrnCallType;
                                ErrorCode : Word) : Word; virtual;
        {-Translate a raw error code into appropriate user error code}
    end;

  function SpecialBiosPrinter.PrnXlatErrorCode(Call : PrnCallType;
                                               ErrorCode : Word) : Word;
  begin
    PrnXlatErrorCode := 0;
    case Call of
      PutCall :
        if ErrorCode and bpSelected = 0 then
          PrnXlatErrorCode := PrinterNotReady;
      StatusCall :
        if ErrorCode and (bpSelected+bpNotBusy) = 0 then
          PrnXlatErrorCode := PrinterNotReady;
    end;
  end;
```

This method performs different status checks depending on which BIOS call was made. PutCall (function 0) tests only the selected bit, and StatusCall (function 2) tests both the selected and not busy bits. This test isn't particularly recommended, although many printers do react differently to the different functions.

Text File Device Drivers

Borland introduced a powerful extension to Turbo Pascal starting with version 4.0 called Text File Device Drivers (TFDDs). A TFDD provides the connection between a read or write to a Text file and the physical input or output operation. By using a TFDD, you can effectively customize the behavior of the Read, ReadLn, Write, and WriteLn statements.

OPPRNLOW provides routines that associate any object derived from BasePrinter with a file of type Text. Thereafter, writes to that file are actually handled by the printer object; all of the flexibility already described for these objects is transferred to the text file.

The following example shows how you would associate the text file variable Lst with a BIOS printer directed to LPT2:

```
program TFDDExample;  {EXPRNLW3.PAS}
uses OpRoot, OpPrnLow;
var  Lst : Text;
     PrnLpt2 : BiosPrinter;
begin
  {Initialize the BiosPrinter object for LPT2}
  if not PrnLpt2.Init(Lpt2) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus); Halt;
  end;
  {Assign the printer object to the text file Lst and open the file}
  AssignPrnObject(Lst, PrnLpt2);
  Rewrite(Lst);
  if IoResult <> 0 then begin
    WriteLn('Error opening TFDD'); Halt;
  end;
  WriteLn(Lst, 'This output goes to LPT2 via BIOS services.');
  Close(Lst);
end;
```

The BiosPrinter PrnLpt2 is bound to the text file Lst. The Lst text file variable would then be used just like the variable of the same name from the Turbo Pascal standard printer unit.

AssignPrnObject takes the place of the Turbo Pascal Assign procedure. It takes two parameters: a file variable of type Text and an already-initialized printer object.

Constants

```
bpTimedOut      = $01;
bpIOError       = $08;
bpSelected      = $10;
bpOutOfPaper    = $20;
bpAcknowledge   = $40;
bpNotBusy       = $80;
```

Bit masks for the BIOS interrupt $17 status byte.

```
PrinterNotReady : Word = 152;
```

Value returned by the PrnXlatErrorCode function, and therefore by IoResult, when a printer error is detected.

```
LptNames : Array[LptType] of String[4] = ('LPT1', 'LPT2', 'LPT3');
```

The names of the three LPT devices.

```
PrnNames : Array[PrnType] of String[12] =
  ('LPT1', 'LPT2', 'LPT3', 'PRN', 'Disk File', 'User Defined');
```

The names of each of the printer types defined by PrnType.

Types

```
PrnType = (LPT1, LPT2, LPT3, Prn, DiskFile, UserDefined);
```

Defines the various types of output devices supported by objects derived from BasePrinter.

```
LptType = LPT1..LPT3;
```

Subset type for BiosPrinters.

```
PrnCallType = (OpenCall, PutCall, CloseCall, FlushCall, StatusCall, ResetCall);
```

Defines the operations common to all BasePrinter-derived objects. A value of this type is passed to PrnXlatErrorCode to indicate the operation which generated the raw error code.

```
XlatCharTable    = Array[Char] of Char;
```

Defines the character translation table.

```
XlatFunc         = function (Call : PrnCallType; ErrorCode : Word) : Word;
```

Function type used to translate raw error codes into logical error codes. See SetXlatErrorFunc for more information.

CharXlat / ClearXlatCharTable / Done

Declaration

```
function CharXlat(C : Char) : Char; virtual;
```

Purpose

Translate the character C according to built-in translation table.

Description

The print output routines call this method just prior to printing each character; an application typically won't call the method directly. The default translation table returns each character unchanged, but the table may be modified using the methods mentioned below.

Derived types may override this function to use a more sophisticated method of character translation than simple table lookup. Another possible reason to override this method would be to trigger an event whenever particular characters were detected. For example, a print spooler could be flushed every time a form feed character was sent to the printer.

Example

```
C := ReadKey;
WriteLn('The character :', C, ' will be translated to :', PrnLpt2.CharXlat(C));
```

Gets a character from the keyboard and displays the raw character followed by the character it will be translated to.

See Also

NewXlatCharTable XlatCharItem

Declaration

```
procedure ClearXlatCharTable;
```

Purpose

Resets the character translation table to default state.

Description

The default state of the translation table is to return each character without translation.

Example

```
PrnLpt2.ClearXlarCharTable;
```

Resets PrnLpt2's translation table to its default state.

See Also

NewXlatCharTable XlatCharItem

Declaration

```
destructor Done; virtual;
```

Purpose

Flush, close, and deallocate memory used by a printer object.

Description

If the printer uses DOS services, then the DOS file or device is flushed and closed. Regardless of the printer type, any heap memory used by the printer object is freed.

Example

```
MyPrinter.Done;
```

Closes the file or device associated with MyPrinter.

See Also

PrnFlush

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a printer from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to DosPrinter.Store. Information that is read includes the port or device name, the character translation table, and status checking information.

The stream registration procedure for a DosPrinter is DosPrinterStream; for a BiosPrinter, BiosPrinterStream. Any user-written functions in use by the object must be registered with a call to the stream's RegisterPointer method before calling Load. A routine activated by SetXlatErrorFunc (BiosPrinter only) is the most likely candidate for this treatment.

If Load fails, InitStatus will contain an error code indicating the cause of the failure.

Example

```
uses OpRoot, OpPrnLow;
var  MyPrinter : DosPrinter;
     PS : BufIdStream;
...
{Load MyPrinter from a stream}
PS.Init('MYPRN.STM', SOpen, 1024);
PS.RegisterHier(DosPrinterStream);
PS.Get(MyPrinter);
if PS.GetStatus <> 0 then begin
  WriteLn('Failed to load printer');  Halt;
end;
PS.Done;
```

This example opens a stream, registers the DosPrinter hierarchy, and calls the stream's Get procedure, which indirectly calls the Load method of DosPrinter. If Load fails, an error is returned by the stream's GetStatus function. If it succeeds, the stream is closed and the program continues.

See Also

BiosPrinter.Init
DosPrinter.Init
Store

Declaration

```
function NameOfPrn : String;
```

Purpose

Returns the printer name.

Description

For DosPrinters, the name is whatever one was used to open the file or device. For BiosPrinters, the name is 'LPT1', 'LPT2', or 'LPT3'.

Example

```
Name := MyPrinter.NameOfPrn;
```

Returns the name of the file or device associated with MyPrinter.

See Also

TypeOfPrn

NewXlatCharTable / PrnError

Declaration

```
procedure NewXlatCharTable(var Table : XlatCharTable);
```

Purpose

Sets the entire XlatChar table.

Description

This routine replaces the object's internal character table with the new one specified.

Example

```
program AllUpper;
uses
  OpRoot, OpCrt, OpPrnLow;
var
  MyPrinter : BiosPrinter;
  UpTable   : XlatCharTable;
  Lst       : Text;
  C         : Char;
begin
  {build a table that translates all characters to upper case}
  DefaultXlatCharTable(UpTable);
  for C := 'a' to 'z' do
    UpTable[C] := UpCase(C);
  {initialize printer object, assign it to a text file}
  MyPrinter.Init(LPT1);
  MyPrinter.NewXlatCharTable(UpTable);
  AssignPrnObject(Lst, MyPrinter);
  Rewrite(Lst);
  {show the results}
  WriteLn(Lst,'abcdefghijklmnopqrstuvwxyz0123456789');
  MyPrinter.Done;
end.
```

See Also

CharXlat
ClearXlatCharTable
XlatCharItem

Declaration

```
function PrnError : Word;
```

Purpose

Returns the last logical error and resets internal status variable to zero.

Description

The code returned is a "logical" error code, compatible with Turbo's IoResult and **Object Professional**'s error code system. To get the raw error code (as discussed for the BIOS status byte), use the PrnRawError function instead.

Example

```
ErrorCode := MyPrinter.PrnError;
if ErrorCode <> 0 then begin
  WriteLn('MyPrinter error = ', ErrorCode);
  Halt;
end;
```

Get the last error code from MyPrinter, store it in a ErrorCode, and display a message and abort if ErrorCode is not zero.

See Also

PrnErrorOp
PrnRawError

Declaration
```
function PrnErrorOp : PrnCallType;
```

Purpose
Returns the type of the last operation.

Description
In the event of an error, it may be useful to know what the printer operation was when the error occurred. See PrnCallType above for more information.

Example
```
ErrorCode := MyPrinter.PrnError;
if ErrorCode <> 0 then begin
  WriteLn('Error code: ', ErrorCode);
  Write('The operation that caused the error was an attempt to ');
  case MyPrinter.PrnErrorOp of
    OpenCall   : Write('open');
    PutCall    : Write('output characters to');
    CloseCall  : Write('close');
    FlushCall  : Write('flush');
    StatusCall : Write('return the status of');
    ResetCall  : Write('reset');
  end;
  WriteLn(' MyPrinter');
end;
```
Writes out the various operations that could have caused the error.

See Also
PrnError

Declaration
```
procedure PrnFlush; virtual;
```

Purpose
Flush the device.

Description
In the case of a DosPrinter, calling this method will flush the file or device by duplicating the file handle and closing the duplicate. For a BiosPrinter, this method does nothing.

Example
```
MyPrinter.PrnFlush;
```
Flush the file or device associated with MyPrinter.

PrnLastPut / PrnPutBlock

Declaration

```
function PrnLastPut : Word;
```

Purpose

Returns the number of characters successfully written by last PrnPutBlock.

Description

This function can be used to implement a error recovery system. In the event of an error on a PrnPutBlock call, determine the number of characters successfully written by calling PrnLastPut, correct the error condition, and resend the characters that were not successfully sent.

Example

```
MyPrinter.PrnPutBlock(S[1], Length(S));
while MyPrinter.PrnError <> 0 do begin
  WriteLn('An error has occurred, please ensure printer is ready');
  WriteLn('Press any key when ready...');
  C := ReadKey;
  Delete(S, 1, PrnLastPut);
  MyPrinter.PrnPutBlock(S[1], Length(S));
end;
```

Checks for an error and retries until all the characters are written.

See Also

PrnError PrnErrorOp

Declaration

```
procedure PrnPutBlock(var Block; BlockSize : Word); virtual;
```

Purpose

Writes a block of characters to the printer.

Description

This method sends BlockSize bytes starting at the address of Block to the output device. If an error is encountered, the method returns immediately. For all provided printer object types, this method calls the PrnPutChar method to send each character.

The compiler provides no error checking on the untyped var parameter passed to PrnPutBlock. Be sure you're printing what you think you are. Typically, the parameter passed for Block will be the first character of a string.

In the event of an error, the method PrnLastPut can be called to determine the number of characters successfully sent before the error occurred.

Examples

```
S := 'Send this string'#13#10;
MyPrinter.PrnPutBlock(S[1], Length(S));
```

This examples assigns a string terminated with a <CR><LF> to the string variable S, then sends the string to the printer.

```
S := 'Send this string'#13#10;
MyPrinter.PrnPutBlock(S, Length(S));
```

Incorrect! This call starts printing with S[0], which is the length byte.

See Also

PrnPutChar

Declaration
```
procedure PrnPutChar(Character : Char); virtual;
```

Purpose
Writes a character to the printer.

Description
The character to write is first translated by calling CharXlat. After the actual call to write, PrnPutChar calls PrnXlatErrorCode to translate the raw error code returned by the output device to a logical error number.

Example
```
MyPrinter.PrnPutChar(^L);
```
Sends the form feed character to the output device.

See Also
PrnPutBlock

Declaration
```
function PrnRawError : Word;
```

Purpose
Returns the raw error code returned by the last operation.

Description
This routine lets you peek at the actual code returned by the last operation. For the BiosPrinter object especially, you could use it to categorize the errors and thereby write more meaningful error messages.

Example
```
var
  MyPrinter : BiosPrinter;
...
if MyPrinter.PrnRawError and bpOutOfPaper <> 0 then begin
  {printer is out of paper}
end;
```
This example shows a test to see if the last error was caused by the printer running out of paper. Not all printers properly set this bit, so the test is not 100% accurate.

See Also
PrnError PrnErrorOp

Declaration
```
procedure PrnReset; virtual;
```

Purpose
Reset the output device.

Description
For the BiosPrinter, this method calls interrupt $17 function 1 to reset the parallel port. For other printer objects, it does nothing. Note that PrnReset doesn't do the same thing as pressing the "Reset" button on a laser printer (which returns the printer to a default font).

Example
```
MyPrinter.PrnReset;
```
Reset MyPrinter's parallel port.

PrnStatus / PrnXlatErrorCode

Declaration

```
function PrnStatus : Word; virtual;
```

Purpose

Return the status of the device.

Description

If this function returns zero, then the device is ready to receive output. Otherwise, there is a problem with the device. PrnStatus first gets the raw status of the device and then calls PrnXlatErrorCode to translate the status to a logical error number. Hence, the meaning of a non-zero error code is dictated by the PrnXlatErrorCode method.

Example

```
while MyPrinter.PrnStatus <> 0 then begin
  WriteLn('Printer is not ready')
  WriteLn('Please prepare printer and press any key (ESC to abort)');
  if ReadKey = #27 then
    Halt;
end;
MyPrinter.PrnPutBlock(S[1], Length(S));
```

Wait until the printer is ready, then write the string S to the output device.

Declaration

```
function PrnXlatErrorCode(Call : PrnCallType; ErrorCode : Word) : Word; virtual;
```

Purpose

Translate raw error code into a logical error code.

Description

BasePrinter methods that can generate an error call PrnXlatErrorCode to translate the raw error into a logical code. DosPrinter's PrnXlatErrorCode method returns the same error number as the DOS call. BiosPrinter's version returns the PrinterNotReady constant if one of the tests described earlier fails. You can override this method to achieve different error handling behavior. PrnXlatErrorCode should always return zero to indicate no error, and a non-zero value that is meaningful to the application to indicate an error.

The Call parameter indicates the type of call that yielded the code to be translated, and the ErrorCode parameter is the code itself.

See the discussion of "Customized Printer Objects" for an example of how to override this routine.

See Also

PrnError

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a printer in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the DosPrinter to the stream S. Information written includes the printer's handle and the character translation table.

To check for errors that may have occurred during the Store, call the stream's GetStatus function.

The stream registration procedure for a DosPrinter is DosPrinterStream, and for a BiosPrinter is BiosPrinterStream. Any user-written functions in use by the object must be registered with a call to the stream's RegisterPointer method before calling Store. A routine activated by SetXlatErrorFunc (BiosPrinter only) is the most likely candidate for this treatment.

Example

```
uses
  OpRoot, OpPrnLow;
var
  MyPrinter : DosPrinter;
  PS : BufIdStream;
begin
  {Open a DosPrinter}
  MyPrinter.Init('PRN', Prn);
  {Open a stream and store the printer to it}
  PS.Init('MYPRN.STM', SCreate, 1024);
  PS.RegisterHier(DosPrinterStream);
  PS.Put(MyPrinter);
  PS.Done;
end.
```

Store the DosPrinter instance MyPrinter in the stream PS.

See Also

BiosPrinter.Init
DosPrinter.Init
Load

TypeOfPrn / XlatCharItem

Declaration
```
function TypeOfPrn : PrnType;
```

Purpose
Returns the type of printer.

Description
This method returns the PrnType of an object: LPT1, LPT2, LPT3, Prn, DiskFile, or UserDefined.

Example
```
WriteLn('MyPrinter is ', PrnNames[MyPrinter.TypeOfPrn]);
```
Uses the predefined constant PrnNames to write out the type of the printer.

See Also
NameOfPrn

Declaration
```
procedure XlatCharItem(CharIndex : Char; Character : Char);
```

Purpose
Change the character CharIndex to Character in the translation table.

Description
After calling XlatCharItem, when the character CharIndex is passed to PrnPutChar or PrnPutBlock, it will be automatically changed to Character.

Example
```
MyPrinter.XlatCharItem('a','A');
S := 'an apple a day';
MyPrinter.PrnPutBlock(S[1],Length(S));
```
XlatCharItem tells MyPrinter to replace all occurrences of 'a' with 'A'. Therefore, PrnPutBlock would print: 'An Apple A dAy'.

See Also
ClearXlatClearTable
NewXlatCharTable
XlatChar

Declaration

```
constructor Init(PrnName : PathStr; TypeOfPrinter : PrnType);
```

Purpose

Initializes a DosPrinter.

Description

The DosPrinter object type uses standard DOS function calls to output the characters. PrnName specifies a file or a device name to open. TypeOfPrinter should be either Prn or DiskFile.

If TypeOfPrinter is Prn, output will be sent to the DOS standard PRN device (handle number 4) without using an additional file handle.

Examples

```
MyPrinter.Init('C:\SPOOL\PRNTEST.DAT', DiskFile);
```

Initializes MyPrinter to write its output to a disk file.

```
MyPrinter.Init('PRN', Prn);
```

Initializes to write to the DOS standard printer device.

```
MyPrinter.Init('LPT2', DiskFile);
```

Initializes to write to LPT2.

See Also

BiosPrinter.Init

GetLptPort / GetTestAndMask

Declaration
```
function GetLptPort : LPTType;
```

Purpose
Returns the LPT port number for this printer.

Description
This method returns either LPT1, LPT2, or LPT3.

Example
```
WriteLn('MyPrinter is using ', LptNames[MyPrinter.GetLptPort]);
```
Uses the predefined constant LptNames to write out the printer port of MyPrinter.

See Also
SetLptPort

Declaration
```
procedure GetTestAndMask(var PrinterTestNumber, Mask : Byte);
```

Purpose
Returns the printer test number and success mask.

Description
See the discussion of "The BiosPrinter Object" in the introduction to this section for more information on printer tests.

Example
```
MyPrinter.GetTestAndMask(TestNo, SuccessMask);
WriteLn('MyPrinter is currently using printer test number ', TestNo);
```
Displays the printer test number currently in use by MyPrinter.

See Also
SetTestAndMask

Declaration
```
function GetXlatErrorFunc : Pointer;
```

Purpose
Returns the XlatError routine currently in use by PrnXlatErrorCode.

Description
This routine is useful if you need to save the current XlatError function, temporarily switch to a new one, and then restore the original.

Because this function returns an untyped pointer, the result must be typecast to an XlatFunc in order to use it as a function of that type.

Example
```
var
  MyPrinter : BiosPrinter;
  SaveXlat : Pointer;
{$F+}
function NewXlatFunc(Call : PrnCallType; ErrorCode : Word) : Word;
begin
  ...
end;
{$F-}
...
SaveXlat := MyPrinter.GetXlatErrorFunc;
MyPrinter.SetXlatErrorFunc(NewXlatFunc);
...
MyPrinter.SetXlatErrorFunc(XlatFunc(SaveXlat));
```

GetXlatErrorFunc returns the current XlatError function and stores it in the untyped pointer variable SaveXlat. The pointer is later typecast to an XlatFunc in order to restore the original error translation function.

See Also
SetXlatErrorFunc

Declaration
```
constructor Init(LPTNumber : LPTType);
```

Purpose
Initialize a BiosPrinter using the best printer ready test.

Description
Init calls InitCustom using a printer ready test of 1. See InitCustom and the discussion of "The BiosPrinter Object" for more information.

Example
```
MyPrinter.Init(LPT3);
```

Initializes MyPrinter to write to LPT3.

See Also
DosPrinter.Init
InitCustom
Load

InitCustom / SetLptPort

Declaration

```
constructor InitCustom(LPTNumber : LPTType;
                       PrinterTestNumber, SuccessMask : Byte);
```

Purpose

Initialize a BiosPrinter with custom options.

Description

LPTNumber specifies which LPT port to use (LPT1, LPT2 or LPT3). PrinterTestNumber indicates which of the provided tests should be used to translate the raw error byte. Valid test numbers range from 0 to 4. SuccessMask is used only for test number 4. See the discussion of "The BiosPrinter Object" for more information.

Examples

```
MyPrinter.InitCustom(LPT2, 2, 0);
```

Initializes MyPrinter to write to LPT2 using test number 2.

```
MyPrinter.InitCustom(LPT1, 4, bpSelected+bpNotBusy);
```

Initializes MyPrinter to write to LPT1 using test 4 and a success mask of bpSelected + bpNotBusy.

See Also

DosPrinter.Init
Init
Load

Declaration

```
procedure SetLptPort(LPTNo : LPTType);
```

Purpose

Sets the LPT port number for this printer.

Description

SetLptPort can change the LPT port for output after the object has been instantiated.

Example

```
var
  MyPrinter : BiosPrinter;
  Lst : Text;
begin
  if not MyPrinter.Init(LPT1) then begin
    WriteLn('Failed to Init, Status = ', InitStatus);
    Halt;
  end;
  AssignPrnObject(Lst, MyPrinter);
  Rewrite(Lst);
  WriteLn(Lst,'This output goes to LPT1');
  MyPrinter.SetLptPort(LPT2);
  WriteLn(Lst,'This output goes to LPT2');
  MyPrinter.Done;
end.
```

Initializes MyPrinter to LPT1 and associates MyPrinter with the text file Lst. The first string is written to LPT1. After SetLptPort is called, the second string is written to LPT2.

See Also

GetLptPort

Declaration
```
procedure SetTestAndMask(PrinterTestNumber, Mask : Byte);
```

Purpose
Set the printer test number and success mask.

Description
SetTestAndMask can change the printer ready test after the object has been instantiated. This is particularly useful while writing an error calibration program. See the discussion of "The BiosPrinter Object" for more information about BIOS printer errors.

Example
```
MyPrinter.SetTestAndMask(2, 0);
```
Set the printer test number to 2. The success mask is ignored for this test.

See Also
GetTestAndMask

Declaration
```
procedure SetXlatErrorFunc(Xlat : XlatFunc);
```

Purpose
Sets the XlatError routine to be called by PrnXlatErrorCode.

Description
The BiosPrinter object normally uses the printer test indicated by the PrinterTestNumber parameter passed to the InitCustom constructor. SetXlatErrorFunc provides a way to switch to a user-written error translation function. Xlat must be a function that is compiled under the FAR model (using {$F+}), is global, and has a declaration that matches the following prototype:

```
{$F+}
function MyXlatFunc(Call : PrnCallType; ErrorCode : Word) : Word;
```

The function should return zero if ErrorCode indicates a successful operation and a non-zero value otherwise.

If you pass NoXlatFunc to SetXlatErrorFunc, BiosPrinter will revert to using the standard printer test.

See "Customized Printer Objects" for more information.

Example
```
{$F+}
function MyXlatFunc(Call : PrnCallType; ErrorCode : Word) : Word;
begin
...
end;
{$F-}
...
MyPrinter.SetXlatErrorFunc(MyXlatFunc);
...
MyPrinter.SetXlatErrorFunc(NoXlatFunc);
```
Transfers error translation to MyXlatFunc and later restores it to the default method.

See Also
BasePrinter.PrnXlatErrorCode GetXlatErrorFunc

AssignPrnObject

Declaration

```
procedure AssignPrnObject(var F : Text; var PrinterObj : BasePrinter);
```

Purpose

Associates the Text variable F with the object PrinterObj.

Description

This method works like the standard Assign procedure, except that it associates a text file with the printing services of the specified printer object. AssignPrnObject works by using a text file device driver.

F is the text file to assign. It should not be open at the time AssignPrnObject is called. PrinterObj is an initialized printer object, of any type derived from BasePrinter. It will probably be of type BiosPrinter or DosPrinter.

AssignPrnObject gives the text file a buffer size of 128 (the default buffer in the Text variable). This size can be increased by calling the standard SetTextBuf procedure. However, additional buffering is of little use since the text file device driver writes each character individually to the printer whenever the buffer is flushed.

After calling AssignPrnObject, you call Rewrite(F) as usual to open the file for output, and then check IoResult.

Example

```
{$I-}
uses
  OpRoot, OpPrnLow;
var
  MyPrinter : BiosPrinter;
  Lst : Text;
begin
  if not MyPrinter.Init(LPT1) then begin
    WriteLn('Failed to Init, Status = ', InitStatus);
    Halt;
  end;
  AssignPrnObject(Lst, MyPrinter);
  Rewrite(Lst);
  if IoResult <> 0 then begin
    WriteLn('Error initializing printer');
    Halt;
  end;
  WriteLn(Lst, 'This output goes to LPT1');
  if IoResult <> 0 then begin
    WriteLn('Error writing to printer');
    Halt;
  end;
  Close(Lst);
  MyPrinter.Done;
end.
```

The BiosPrinter object MyPrinter is associated with the text file Lst by calling AssignPrnObject. Once Rewrite has been called successfully, the example can write to the Lst file just like usual. In this example, the final calls to Close and Done are not essential, but using them is always good methodology.

See Also

BiosPrinter.Init DosPrinter.Init

Declaration

```
function CallPrnStatus(var F : Text) : Word;
```

Purpose

Return the result of the PrnStatus method for the printer object associated with F.

Description

If the text file F does not have a printer object associated with it, CallPrnStatus returns $FFFF. CallPrnStatus differentiates between normal text files and those that are attached to printer objects.

Use this method to check for printer ready before attempting to write.

Example

```
var
  Lst : Text;
  LstPrn : BiosPrinter;
...
if not LstPrn.Init(LPT1) then
  Halt;
AssignPrnObject(Lst, LstPrn);
Rewrite(Lst);
if IoResult <> 0 then
  Halt;
if CallPrnStatus(Lst) = 0 then
  WriteLn(Lst, 'This string goes to the printer')
else
  WriteLn('Printer is not ready');
```

Opens a Text file using BiosPrinter services and checks whether the printer is ready before writing to it.

See Also

AssignPrnObject

BasePrinter.PrnStatus

DefaultXlatCharTable / GetPrnObject

Declaration

```
procedure DefaultXlatCharTable(var Table : XlatCharTable);
```

Purpose

Initialize Table to default state.

Description

After calling this routine, Table will contain the identity mapping, that is, a translation will return each character unchanged. This routine is called by the BasePrinter methods NewXlatCharTable and ClearXlatTable.

See Also

BasePrinter.ClearXlatTable
BasePrinter.XlatChar
BasePrinter.XlatCharItem

Declaration

```
function GetPrnObject(var F : Text) : BasePrinterPtr;
```

Purpose

Return a pointer to the printer object associated with a text file.

Description

This function returns nil if the text file does not have a derivative of a BasePrinter object instance associated with it. This routine is primarily for internal use.

Example

```
var
  P : BasePrinterPtr;
...
P := GetPrnObject(PrnTextFile);
if P = nil then
  WriteLn('No printer object associated with PrnTextFile')
else
  WriteLn('PrnTextFile = ', P^.NameOfPrn);
```

Gets a pointer to the object associated with PrnTextFile. If the returned pointer is not nil, then the name of the print device is displayed.

See Also

CallPrnStatus

C. OPPRINT: Logical Level Printer Support

Whereas OPPRNLOW focused on transferring characters to the printer and reading back the status, OPPRINT deals with the appearance of printed characters on the page. OPPRINT's primary object, Printer, provides the power needed to change fonts, enable alternate printing modes, modify lines per inch, draw lines and boxes–basically to take advantage of all the features of any printer.

The Printer object doesn't know a thing about any particular printer device. An instance of a Printer object must be initialized to understand those details. It's the OPDEVICE unit's responsibility to know about specific devices; we'll describe OPDEVICE in the next section.

(Note that we capitalize Printer when we mean the object type, and use lower case when we refer to the noisy box that plugs into the wall.)

In the best of all worlds, OPDEVICE would know everything there is to know about 1000 different printers. You'd just tell OPDEVICE to register a Printer object as an "Epson 4000ZX" printer, ask for red-on-green helvetica bold in 24 point, and start blazing away. In this case, you wouldn't need to know much about the Printer object except how to initialize it, how to ask for a particular print mode, and how to print a string. Well, we're not quite to nirvana yet. If the printers you want to support are on the list of the most common--Epson dot matrix, H-P LaserJet, Diablo daisy-wheel, and a growing number of others–then you'll be able to ignore many of the details of the Printer object. If you need to support other devices or take advantage of rarely used printer features, then you'll need to learn more about Printer. In either case, you'll find it worthwhile to continue reading this introduction.

Here is a list of terms that we will use frequently throughout the rest of this chapter.

Backspace
: The ability of a printer to back up by one or more character positions on the same line. Usually activated by a backspace character (ASCII character 8, <CtrlH>).

BasePrinter
: Any object derived from the BasePrinter type defined in OPPRNLOW; used as shorthand for saying "BiosPrinter, DosPrinter, or perhaps a FlexiblePrinter."

Control sequence
: A sequence of characters sent to a printer that cause it to perform a command such as switching modes, changing fonts, or repositioning the printhead. The characters themselves do not appear on the page.

Double strike
: Enhanced print intensity achieved by backspacing and printing the same character two or more times. Some printers support a special mode that performs this operation automatically and also offsets the second strike by a small distance.

Characters per inch
: The number of characters that will fit horizontally in an inch. Abbreviated CPI.

Character pitch
: The same as characters per inch, an inverse measure of character width.

Character point
: The height of a character measured in points.

CR/LF
: An abbreviation for carriage return (ASCII character 13, <CtrlM>) followed by line feed (ASCII 10, <CtrlJ>). Interpreted by most printers to mean "advance printhead to beginning of next line."

Decipoint
A unit of measurement that equals 1/720 of an inch.

Escape sequence
Same as control sequence.

Font
A particular character printing style and size, e.g., bold 12 point Roman. Unlike real typesetting experts, we won't distinguish between fonts and typefaces.

Form feed
A printer command that means "advance to top of next page." Also a character (ASCII 12, <CtrlL>). Abbreviated FF.

Landscape mode
A page orientation where the horizontal direction is parallel to the long dimension of the paper–"sideways printing." The opposite of portrait mode.

Leading
The distance between two rows of characters. Note that the character height (character point) and the leading are not necessarily the same. Leading is the inverse of lines per inch. (Pronounced "ledding")

Lines per inch
The number of rows of characters that will fit in an inch. Abbreviated LPI.

Lines per page
The number of rows of characters that will fit on a page. Abbreviated LPP.

Point
A unit of measurement that equals 1/72 of an inch, usually referring to character height. 12 point characters are usually printed at 6 lines per inch.

Portrait mode
A page orientation where the horizontal direction is parallel to the narrow dimension of the paper–"normal printing." The opposite of landscape mode.

Positionable printer
A printer that supports easy positioning of the printhead anywhere on the page in any sequence–typically a laser printer. Some non-laser printers support reverse line feeds; OPPRINT could be forced to treat these as positionable, but we don't recommend it.

Printer intensity
The darkness of the printed output. May be achieved by modes built into a printer or by doublestriking.

Printhead position
The location on a page where the next output will be printed. For a laser printer, there is no physical printhead, but the concept of printhead position is still applicable.

Print modes
Refers generically to the capabilities of a particular printer, e.g., its fonts, orientations, and line spacings.

Reverse linefeed
The ability of a printer to back up by one or more lines on the same page.

Rule
A horizontal or vertical line drawn by a printer.

The PrintMode Object

PrintMode is an object that encapsulates a pair of printer control sequence strings: one to activate a particular mode and another to deactivate it. You typically won't declare instances of the PrintMode type since the Printer object allocates and manages them for you. Even so, the PrintMode is the basis for much of Printer's flexibility, so it's important to know its capabilities.

Each PrintMode has an identifying code (a byte) and an optional identifying name (a string) that allow it to be easily found within a linked list of PrintModes. It also contains a bit-mapped option word that describes the effects of the mode (whether it changes LPI, changes margins, changes page orientation, and so on). PrintMode also contains width and height fields that describe its numeric effects on LPI, CPI, and so on.

Since access to PrintMode objects occurs strictly through methods of Printer, we won't document any of its methods. However, see the Constants section below for more details about PrintMode options.

The Printer Object

The Printer object is a large one, with over 50 documented methods and another 70 or so for internal use. If the printer you wish to access is already supported by OPDEVICE, the apparent complexity of Printer drops immediately, since many of the methods are intended for initialization of specific printer characteristics, and OPDEVICE takes care of that for you. The remaining methods are used primarily for enabling and disabling print modes, positioning the printhead, and writing strings.

In this section, we'll provide an overview of the Printer object: its data fields and its methods. To give you a feeling for how it's used, let's start with an example.

```
program PrinterExample;  {EXPRINT.PAS}
uses OpRoot, OpPrnLow, OpPrint;
var  MyPrinter : Printer;
begin
  if not MyPrinter.Init('PRN', Prn, 0) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus);
    Halt;
  end;
  MyPrinter.AddPrintMode('Italics', #27'4', #27'5', 0, 0, Italics, 0);
  MyPrinter.TurnOnByID(Italics);
  MyPrinter.PrintStr('Hello World');
  MyPrinter.Position(50, 1, rtAbsolute);
  MyPrinter.TurnOffByID(Italics);
  MyPrinter.PrintStr('Goodbye World');
  MyPrinter.FormFeed;
  MyPrinter.Done;
end.
```

The example instantiates a Printer object that contains a DosPrinter (from OPPRNLOW) to write to the PRN device. A PrintMode defining the (Epson) control sequences for italics is added. Italics are turned on and a string is written to the current page position. The printhead is moved toward the bottom of the page, italics are turned off, and another string is printed. A FormFeed is then issued to eject the page, and the Done destructor is called.

The nice thing about this example is that only the AddPrintMode call is specific to a particular printer. One of the main goals of the Printer object is to isolate the methods for printing strings from the methods for initializing a specific hardware device.

The Printer object refers to a number of data fields and parameters by using the data type "Dimension." A Dimension is the measure of a coordinate position on the printed page. The term is deliberately vague, because the Printer object doesn't restrict how these distances are to be measured. Dimension is a synonym for a six byte Real data type, or a Double if the {$N+} compiler option is set. Hence, variables of type Dimension may be specified with arbitrary accuracy.

By default, OPPRINT, OPDEVICE, and OPFORM treat most variables of type Dimension in units of rows and columns. For example, Printer's method to write a string at a specified position is declared as:

```
procedure PrintStrPos(S : String; Row, Col : Dimension;
                      Relative : RelativeType); virtual;
  {-Print a string at the specified position}
```

While executing the PrintStrPos method for a non-positionable printer, OPPRINT generates a sequence of CR/LF's and spaces or backspaces to reach the specified Row and Col. Non-integer values of Row and Col are rounded to the nearest integer value. For a supported positionable printer like the LaserJet, all values of Row and Col are converted into an appropriate escape sequence, which positions the printhead at the specified position to the full resolution of the printer.

OPPRINT provides another level of flexibility for defining printhead coordinates. Before acting on any positioning command, OPPRINT multiplies the user-specified coordinates by internal X and Y scaling ratios. These factors default to 1.0 but can be set to any value. By using these ratios you could effectively redefine the units of "Row" and "Col" to inches or centimeters.

Among the Printer's data fields are

- a bit-mapped printer options word
- an error status variable
- a code identifying the registered printer type
- the last known position of the printhead
- the X and Y scaling ratios
- the current values of LPI, CPI, LPP, and character height
- the ID code and address of the current PrintMode
- the address of the associated BasePrinter object
- a linked list of associated PrintMode objects
- a ByteStack of the ID codes of active fonts
- a BitSet of the ID codes of all active PrintModes
- the number of strikes for double strike mode
- procedure and function pointers to support advanced capabilities like random printhead positioning and user-written error handlers

You can read and write these fields, when necessary, by calling appropriate methods of the Printer object.

The methods of the Printer fall into several categories, which are listed here in the approximate order that you'll call them in a program. After each category are the names of the related methods in order to make it easier for you to use the alphabetically organized method descriptions later in this section:

Constructors

```
Init, InitCustom
```

Modifying and retrieving data fields

```
SetLPP, SetYRatio, SetXRatio, SetRegisteredType, pOptionsOn, pOptionsOff,
GetLPP, GetYRatio, GetXRatio, GetRegisteredType, pOptionsAreOn, pGetLastError
```

Linking to and accessing a BasePrinter object

```
LinkToPrinter, GetBasePrinter, PrinterReady
```

Adding new print modes
AddPrintMode, LinkToModes, DisposeAllModes

Activating user-written hook routines
SetPrintErrorFunc, SetPositionPrim

Positioning the printhead
Position, FormFeed, AdvanceLine, AdvanceSingleLine, ResetPage

Selecting and accessing print modes
SetDefaultFont, EnableDefaultFont, TurnOnByID, TurnOffByID, TurnOnByName, TurnOffByName, TurnOffAllActiveModes, GetCurrentFont, GetDefaultFont, GetPrintModeByID, GetPrintModeByName, InLandscape

Controlling double strike mode
EnableDoubleStrike, DisableDoubleStrike, SetStrikeCount, InDoubleStrike, StrikeCount, CurrentStrikes

Printing strings
PrintStr, PrintStrPos

Destructor
Done

Stream Support
Load, Store

Several of Printer's data fields and methods are used for informational purposes only, i.e., the fields can be updated and read, but Printer doesn't use them directly to modify what appears on the printed page. Data fields for LPI, CPI, character height, and top margin fall into this category. (OPDEVICE refers to LPI and CPI in implementing decipoint support for the LaserJet.) When you turn on a PrintMode that has the appropriate flags set, these data fields are updated to reflect the values stored for that mode. Your program can then refer to them in order to perform position calculations of its own. The following methods fall into this category and are not documented in detail. See AddPrintMode for further information.

```
procedure SetPrinterInfo(LPI, CPI, Point, LPP : Dimension);
  {-Sets lines per inch, chars per inch, character height, and lines per page}
procedure SetLPI(LPI : Dimension);
  {-Sets lines per inch}
procedure SetCPI(CPI : Dimension);
  {-Sets chars per inch}
procedure SetPoint(Point : Dimension);
  {-Sets the character point}
procedure SetPhysicalTopMargin(Top : Dimension);
  {-Sets the physical top margin}
procedure GetPrinterInfo(var LPI, CPI, Point, LPP : Dimension);
  {-Returns lines per inch, chars per inch, character height, and lines per page}
function GetLPI : Dimension;
  {-Returns lines per inch}
function GetCPI : Dimension;
  {-Returns chars per inch}
function GetPoint : Dimension;
  {-Returns the character point}
procedure GetPhysicalTopMargin(var Top : Dimension);
  {-Returns the physical top margin}
```

Fonts and the Font Stack

As far as the Printer object is concerned, a font is usually just another kind of PrintMode. Of course, to you a font is one of the most important characteristics of the printed output; you need bold, italic, compressed, or other kinds of type.

The Printer object often must take additional steps to enable and disable fonts compared to what it does for some other PrintModes. For example, if you are printing in an italic font, then enable a bold font and print a string, and finally disable the bold font, you probably expect the next output to appear in italics. Some printers have this kind of behavior built in. For example, many of the "fonts" on an Epson dot matrix are really characteristics that can be toggled on with one control sequence and off with another. However, other printers may not have a control sequence that turns *off* a particular font; hence, the Printer object must be smart enough to re-enable a previous font when you ask to turn off the current one.

To deal with fonts that don't work like the Epson toggles, Printer manages a stack of active fonts. Whenever you activate a particular PrintMode, Printer must perform the follow actions:

- store the address of the new PrintMode in a Printer data field
- handle the special cases of double strike and printer reset
- if the PrintMode is a non-toggle font, push its ID code onto the font stack
- write the enabling control sequence to the printer
- update a BitSet to indicate that this particular mode is enabled
- update internal Printer data fields affected by the PrintMode

When you deactivate a PrintMode, Printer can then use the stack to reenable the previous font:

- handle the special case of disabling double strike
- write the disabling control sequence, if any, to the printer
- update a BitSet to indicate that this particular mode is disabled
- if the disabled PrintMode is a non-toggling font, pop the stack and activate the PrintMode whose ID is now at the top of the stack

Several caveats and restrictions apply to this algorithm. First, it assumes that you eventually ask to deactivate each non-toggling font that you activated originally. If instead you just activate each font as you need it, the font stack will eventually overflow, generating a program error. While overflow is unlikely (the font stack can hold FontStackSize–255 by default–font ID codes), you wouldn't want to use this approach to print long sequences of formatted output on certain printers. Second, if you ask to deactivate a font that's not the last one activated, you may not get the results you expect. If the deactivated font is not on top of the stack, Printer scans the remainder looking for the most recently pushed matching font. If it finds a match, it removes that element from the stack and re-activates the font at the top of the stack. If it doesn't find a matching font, it generates a warning status code. The warning activates the printer error handler if one is installed. If the handler returns True, then the default font, if any, is enabled; otherwise, no further action is taken and printing continues.

Not all fonts need to be pushed on the font stack. Only PrintMode objects that have the pmFont flag set are pushed. When you add a PrintMode you can generally decide whether to set the pmFont flag by noting whether the font has an unambiguous control sequence available to disable the font. If there is no such control sequence, or if the control sequence will potentially disturb the previously active font, the pmFont flag is called for.

The special case of double strike is discussed in the section entitled "Double Strike Mode" below. The other special case, ResetPrinter, causes the font stack to be cleared and never reenables the default printer attribute.

You'll also note the mention of a BitSet in the algorithm outline. OPPRINT maintains this set (of active PrintMode ID numbers) so that TurnOffAllActiveModes can deactivate all active modes. The BitSet tracks up to pMaxPrintModes (255) unique print modes per Printer object.

Laser Printer Support

OPPRINT includes several options that are designed to support special features of printers like the Hewlett-Packard LaserJet. The simplest is landscape orientation. An appropriate PrintMode can switch the printer's mode and set an internal flag that Printer will use to detect landscape mode. See the AddPrintMode method for more information.

LaserJet printers support a positioning interval called the decipoint (1/720 of an inch). If the Printer option pUseDecipoints is set, row and column coordinates passed to Printer methods will be interpreted as decipoints rather than as units of character height and width.

The LaserJet offers special commands for drawing vertical and horizontal lines of arbitrary width and position. The LaserJet goes even further by supporting pattern fills and shading commands. If a printer supports these LaserJet capabilities, the pSupportsHPRules Printer option flag should be set. Although OPPRINT doesn't interpret this option, OPFORM does, and you'll be able to use its LineField object to create impressive laser-printed forms.

Double Strike Mode

If the previous section can be said to describe the printers of the future, this one describes the dinosaurs of the past. Daisy wheel printers and some very old dot matrix printers create bold print only by backspacing and reprinting each character. The Printer object goes to some lengths to support this mode of operation. Although this mode was added to aid primitive printers, it may also be useful for intensifying the fonts of more advanced printers or for printing on multi-part forms. Note, however, that backspacing and reprinting on a laser printer has no visible effect, since the same bits are set within the printer's page buffer each time.

Although "double strike" leads one to think of backspacing and reprinting a single time, OPPRINT allows you to specify up to 127 overprints. The SetStrikeCount method changes this count from its default value of 2. StrikeCount returns the strike count most recently assigned.

EnableDoubleStrike sets an internal flag that directs OPPRINT to print each character StrikeCount times. DisableDoubleStrike resets this flag. InDoubleStrike returns True if double striking is enabled.

While double striking is active, do not print strings that contain carriage returns, line feeds, or other characters whose effects cannot be undone by writing a backspace. Instead use AdvanceSingleLine, FormFeed, TurnOnByID, and other methods to print these special characters; none of these methods are affected by double strike mode.

When you call TurnOnByID with an ID code of DoubleStrike, OPPRINT generates a call to EnableDoubleStrike instead of sending a control sequence to the printer. Conversely, TurnOffByID with the same ID code calls DisableDoubleStrike.

With that said, the following Printer methods are obvious enough that they need no further mention in the method reference section that follows:

```
procedure EnableDoubleStrike;
  {-Enable double strike mode}
```

```
procedure DisableDoubleStrike;
  {-Disable double strike mode}
function InDoubleStrike : Boolean;
  {-Returns True if double strike mode is active}
function StrikeCount : ShortInt;
  {-Returns the number of strikes per character for double strike mode}
procedure SetStrikeCount(Strikes : ShortInt);
  {-Set the number of strikes per character for double strike mode}
```

Printer Error Handling

Printer-related errors are often temporary conditions that can be corrected. To take advantage of this fact, you install an error handler that OPPRINT calls whenever an error occurs. OPPRINT informs the handler whether the error appears to be correctable; the handler prompts the operator to correct the problem; if the operator complies, the handler returns with a flag set for OPPRINT to retry the operation.

Without such a handler, when an error occurs, the Printer object simply stores an error code in an internal variable and treats the error as non-correctable. The operation that caused the error is either ignored or aborted. The application program must call the pGetLastError method after each call to a Printer method.

You can install your own error handler by using the SetPrintErrorFunc method:

```
procedure SetPrintErrorFunc(PEF : PrintErrorFunc);
  {-Set the print error function}
```

PEF is a function that must be compiled under the FAR model (using the {$F+} compiler directive), must be global, and must have parameters that match the following prototype declaration (defined by the OPPRINT type PrintErrorFunc):

```
{$F+}
function MyPrintErrorFunc(P : PrinterPtr; ErrorCode : Word;
                          Recoverable : Boolean) : Boolean;
```

When OPPRINT calls this routine, it passes the address (P) of the Printer instance that encountered the error, an error code (ErrorCode) that is consistent with those described in Appendix D, and a boolean (Recoverable) that indicates whether OPPRINT considers the error to be correctable. For appropriate values of ErrorCode and Recoverable, the error handler should display an error message and prompt the user to correct the situation. When the user indicates that the problem has been corrected, the error handler should return True to trigger a retry, or False to abort the operation.

Here's a simple example of an error handler:

```
{$F+}
function MyPrintErrorFunc(P : PrinterPtr; ErrorCode : Word;
                          Recoverable : Boolean) : Boolean;
begin
  MyPrintErrorFunc := False;
  WriteLn('A printer error has occurred, the error code is ', ErrorCode);
  if Recoverable then begin
    WriteLn('Please correct the problem');
    WriteLn('Press any key when ready, or <Esc> to abort');
    MyPrintErrorFunc := (ReadKey <> #27);
  end else
    WriteLn('Operation aborted');
end;
```

Note that OPPRINT keeps track of exactly how many characters of a string have been printed. After a corrected error it starts printing again with the particular character for which it detected an error. Therefore, if output is going to a non-buffered printer, complete error recovery is possible. However, if the printer itself has a buffer or if a print spooler is involved, OPPRINT may not learn of the error until long after it has sent additional characters. In this situation, it is the buffer's responsibility to retry appropriately. Buffers aren't usually very good about this. It's also very easy to start printing on a poorly aligned form or to have a paper jam. As a result, programs that are designed for serious printing should always have an option to abort a given printout and start over.

The PrinterPtr parameter is provided in case more specific data is required from the Printer object. For example, suppose you wanted the raw error code collected by the underlying BasePrinter object; the following example shows how to get it.

```
with P^ do
  if TypeOf(GetBasePrinter^) = TypeOf(BiosPrinter) then
    if GetBasePrinter^.PrnRawError and bpOutOfPaper <> 0 then
      {Printer is out of paper};
```

If the BasePrinter is an instance of a BiosPrinter, then the out of paper bit is checked in the raw error code returned by its PrnRawError method.

Printer Objects and Streams

As with most objects in **Object Professional**, those in OPPRINT can be stored in streams. Streams are particularly useful with the Printer object, since the Printer's image in the stream includes all of the PrintMode control sequences and other nitty-gritty details about the printer. The OPDEVICE unit's documentation discusses this issue in greater detail.

Constants

```
pmFont              = $0001;
pmChangesLPI        = $0004;
pmChangesLPP        = $0008;
pmChangesCPI        = $0010;
pmChangesPoint      = $0020;
pmChangesTopMargin  = $0040;
pmChangesLandscape  = $0080;
pmChangesStrike     = $0100;
```

These are the options that apply to each PrintMode added to a Printer. One or more of these values are passed to the AddPrintMode method, which contains more information about all of these options. pmFont indicates that the mode refers to a non-toggling font (e.g., bold roman on a LaserJet) and therefore must be treated as described in the section "Fonts and the Font Stack" above. pmChangesLPI, pmChangesLPP, pmChangesCPI, pmChangesPoint, and pmChangesTopMargin indicate that the mode changes the indicated characteristic of the page or character dimensions. The appropriate dimensional value from the PrintMode object will update the current value of LPI, LPP, CPI, Point, or TopMargin in the Printer object when such a mode is turned on. pmChangesLandscape indicates that the mode changes the printer orientation, which again updates a flag internal to the Printer object when enabled. pmChangesStrike indicates that the mode changes the double-strike behavior of the printer.

```
pPlain = 0;
```

A hardware registration number that means no particular printer device type has been registered to this Printer. Additional registration numbers and routines for specific printers are available within the OPDEVICE unit.

```
FontStackSize       = 255;
```

Maximum number of fonts that can be pushed on the font stack.

```
pMaxPrintModes      = 255;
```

Maximum number of print modes that can be associated with a given printer.

```
pPositionable       = $0001;
pBackSpaceAllowed   = $0002;
pNoFormFeed         = $0004;
pSoftFormFeed       = $0008;
pSupportsHPRules    = $0010;
pUseDeciPoints      = $0020;
```

These option flags apply to a Printer as a whole. pPositionable indicates that the physical printer's internal cursor can be positioned randomly on the page; this generally applies to laser printers only. pBackSpaceAllowed indicates that the printer interprets ^H as a command to move the printhead one column to the left. If pNoFormFeed is enabled, Printer ignores calls to the FormFeed method; this may be useful when generating draft versions of reports. If pSoftFormFeed is enabled, Printer ejects a page by printing enough CR/LF's to fill out the page; otherwise it just writes a form feed character. pSupportsHPRules indicates that the printer supports the Hewlett-Packard LaserJet line and box drawing commands. This option enables extra functionality in the OPFORM unit. If pUseDeciPoints is set and a LaserJet printer is registered (see OPDEVICE), coordinates passed to Position and PrintStrPos methods are interpreted in decipoints rather than in rows and columns.

```
DoubleStrike        = 255;
PrinterReset        = 254;
Italics             = 253;
Bold                = 252;
Dim                 = 251;
Compressed          = 250;
Underline           = 249;
Unidirectional      = 248;
DoubleWidth         = 247;
LetterQuality       = 246;
SelectPrimaryFont   = 245;
SelectSecondaryFont = 244;
DefaultFont         = 243;
```

Standard identification codes associated with PrintMode objects. It's recommended that you pass these constants to methods like TurnOnByID and TurnOffByID rather than referring to specific numbers. DoubleStrike is interpreted as a special case: when enabled, OPPRINT backspaces and reprints the characters a specified number of times; see the discussion of "Double Strike Mode" above. PrinterReset is also treated as a special case when enabled; besides sending the control sequence it also clears the font stack as described in "Fonts and the Font Stack" above. Not all printers may support all of these modes, and OPDEVICE does not add print modes for every one. You may assign additional codes in the range 1..242.

```
DefaultStrikes : Byte = 2;
```

The default number of strikes in double strike mode.

```
DefaultLPI : Dimension = 6.0;
```

The default number of lines per inch used by Printer.Init.

```
DefaultCPI : Dimension = 10.0;
```

The default number of characters per inch used by Printer.Init.

```
DefaultPoint : Dimension = 12.0;
```

The default character point used by Printer.Init.

```
DefaultLPP : Dimension = 66.0;
```

The default number of lines per page used by Printer.Init.

```
BackSpaceChar : Char = #8;
```

The character used to backspace the printhead.

```
LineAdvanceStr : String[11] = ^M^J;
```

The string used to advance to the beginning of the next line.

```
FormFeedStr : String[11] = ^L;
```

The string used to issue a form feed.

Types

```
{$IFOPT N+}
Dimension = Double;
{$ELSE}
Dimension = Real;
{$ENDIF}
```

The data type used to store all printhead coordinates, height, and width data within the PrintMode and Printer object types. This type is also used by the OPDEVICE and OPFORM units. If the N+ compiler directive is in effect, then Dimension will be a synonym for type Double, otherwise Dimension will be a synonym for type Real.

```
RelativeType = (rtAbsolute, rtCurrent);
```

The two types of positioning available: rtAbsolute means that specified Row and Col coordinates are interpreted with respect to the upper left corner of the page; rtCurrent means that they are relative to the last known position of the printhead. Note that, unless the Printer's pPositionable flag is set, the printhead cannot back up vertically.

```
PrintErrorFunc = function(P : PrinterPtr; ErrorCode : Word;
                          Recoverable : Boolean) : Boolean;
```

Function type for a printer error handler. See "Printer Error Handling" above for more information.

```
PrinterPosFunc = function(Row, Col : Dimension; Relative : RelativeType;
                          P : PrinterPtr) : Word;
```

Function type definition for the primitive routine used to position the printhead. See SetPositionPrim for more information.

```
PrintModePtr = ^PrintMode;
PrintMode =
  object(SingleListNode)
    ...
  end;
```

The PrintMode object type. This object is not documented in the following. See OPPRINT.PAS for more information.

```
PrinterPtr = ^Printer;
Printer =
  object(Root)
    ...
  end;
```

The Printer object type. All relevant data fields are accessible via methods described in the following section.

AddPrintMode

Declaration

```
procedure AddPrintMode(Name, OnEscSeq, OffEscSeq : String;
                       Width, Height : Dimension; ID : Byte; OptionFlags : Word);
```

Purpose

Add a Print Mode to a Printer's linked list of modes.

Description

Name specifies an ASCII name for the mode. GetPrintModeByName uses this string to search for a print mode. Name may be an empty string if you don't care to search this way. GetPrintModeByID works even if Name is empty.

OnEscSeq is the control sequence used to enable this printer mode, i.e, the string sent to the printer when TurnOnByID is called. OffEscSeq is the control sequence used to disable the option, i.e., the string sent by TurnOffByID. AddPrintMode allocates just enough heap space to hold each string. Either string may be empty.

The Width and Height parameters are relevant only if one of the pmChangesXxx options is specified in OptionFlags. They are interpreted as shown in the following table:

OptionFlag	Width	Height
pmChangesLPI	--	lines per inch
pmChangesLPP	--	lines per page
pmChangesCPI	characters per inch	--
pmChangesPoint	--	character height
pmChangesTopMargin	--	top margin
pmChangesLandscape	--	0 for portrait, 1 for landscape
pmChangesStrike	--	strike count (1 or greater)

When the PrintMode is enabled, the appropriate value stored in Width or Height updates the internal data fields of the Printer object. In most cases only one OptionFlag bit may be specified per call to AddPrintMode to avoid running into conflicts with the interpretation of Width and Height.

ID is a value that uniquely identifies this PrintMode (see the Constants section above for predefined ID values).

OptionFlags specifies what kind of PrintMode this one is (see the Constants section above for further definitions pertinent to this parameter). It is common to specify zero for this parameter.

If there is insufficient memory to allocate another PrintMode, AddPrintMode generates an error (epNonFatal + ecOutOfMemory) and exits. If pMaxPrintModes (255) modes have already been added, AddPrintMode generates another non-fatal error. Errors may be checked by calling pGetLastError.

Example

```
var MyPrinter : Printer;
...
MyPrinter.AddPrintMode('Italics', #27'4', #27'5', 0, 0, Italics, 0);
...
MyPrinter.TurnOnByID(Italics);
MyPrinter.PrintStr('Hello World');
```

Adds a print mode named 'Italics' to MyPrinter. For this printer (an Epson MX compatible) italics are turned on by <Esc>4 and off by <Esc>5. No width or height parameters are required for this particular option. The Italics constant is later used to refer to this PrintMode. Note that this print mode is added automatically by the EpsonMXRegister procedure discussed in OPDEVICE.

See Also

TurnOffByID
TurnOffByName
TurnOnByID
TurnOnByName

Declaration
```
procedure AdvanceLine(NumLines : Byte);
```

Purpose
Advances printhead by NumLines number of lines.

Description
This routine makes NumLines calls to the virtual method AdvanceSingleLine.

Example
```
MyPrinter.AdvanceLine(5);
```
Advances printhead by 5 lines.

See Also
FormFeed

Declaration
```
procedure AdvanceSingleLine; virtual;
```

Purpose
Advances the printhead to the beginning of the next line.

Description
This method writes LineAdvanceStr (which defaults to CR/LF) to the printer and then updates internal variables that indicate the printhead's current row and column coordinates.

This routine is virtual so that derived types may gain control over how the printhead is advanced to the next line.

Example
```
MyPrinter.AdvanceSingleLine;
```
Advances printhead to the next line.

See Also
FormFeed

Declaration
```
procedure DisposeAllModes;
```

Purpose
Disposes of all print modes associated with this Printer.

Description
This method provides a quick means of getting rid of all print modes associated with a Printer object without disposing of the object itself. After calling this method, new print modes may be added by calling AddPrintMode or by calling one of the high level hardware registration routines (documented in OPDEVICE).

Example
```
MyPrinter.DisposeAllModes;
```
Disposes of all of MyPrinter's print modes.

See Also
AddPrintMode

Done / EnableDefaultFont

Declaration

```
destructor Done; virtual;
```

Purpose

Destroy a Printer object.

Description

This method should be called when you are finished using a Printer object instance. This method shuts down the printer, freeing all associated memory.

If the Printer object allocated the associated BasePrinter object itself (which occurs if the Init constructor was called), then Done calls the BasePrinter's Done method and deallocates the BasePrinter. If the BasePrinter was not allocated by Init (which is the case if InitCustom was called), Done does not close or dispose of the BasePrinter.

Examples

```
var
  MyPrinter : Printer;
...
  MyPrinter.Done;
```

Shut down the static Printer object MyPrinter.

```
var
  MyPrinterPtr : PrinterPtr;
...
  Dispose(MyPrinterPtr, Done);
```

Shut down and dispose of the dynamic instance of a Printer.

See Also

Init
InitCustom
LinkToPrinter
Load

Declaration

```
procedure EnableDefaultFont;
```

Purpose

Enables the default font.

Description

The default font ID is assigned by the SetDefaultFont method and returned by GetDefaultFont. EnableDefaultFont does the equivalent of TurnOnByID(GetDefaultFont). If no default font has been assigned, GetDefaultFont returns zero and EnableDefaultFont does nothing.

OPPRINT sometimes calls EnableDefaultFont automatically if it pops the font stack (see "Fonts and the Font Stack," above) and finds that the stack is empty.

This method is used primarily by the hardware registration routines in OPDEVICE.

Example

```
MyPrinter.EnableDefaultFont;
```

Enables the default font associated with MyPrinter.

See Also

GetDefaultFontTurnOnByID
SetDefaultFont

Declaration
```
procedure FormFeed;
```

Purpose
Advance to next page.

Description
If the pNoFormFeed option is enabled, FormFeed does nothing. If pSoftFormFeed is set, FormFeed writes CR/LF's to fill out the current page (based on the current value of lines per page and the current printhead position). If neither option is set, FormFeed writes FormFeedStr, which defaults to <CtrlL>.

In all of these cases, FormFeed calls ResetPage to reset the Printer object's internal record of the printhead position.

Example
```
MyPrinter.FormFeed;
```
Advance to the top of the next page.

See Also
AdvanceLine ResetPage

Declaration
```
function GetBasePrinter : BasePrinterPtr;
```

Purpose
Returns the BasePrinterPtr associated with a Printer.

Description
Each Printer object contains a pointer to an associated BasePrinter object. GetBasePrinter returns a pointer to that object, thus allowing you to access any of its methods or data. See the OPPRNLOW section for more information.

Example
```
MyPrinter.GetBasePrinter^.PrnReset;
```
Calls the PrnReset method of the BasePrinter associated with MyPrinter.

See Also
LinkToPrinter

Declaration
```
function GetCurrentFont : Byte;
```

Purpose
Returns the current font's PrintMode ID, or zero if no font is currently in use.

Description
Whenever a non-toggling font is enabled (by calling TurnOnByID or TurnOnByName), its ID is pushed onto the font stack. GetCurrentFont returns the ID at the top of the font stack.

Example
```
FontInUse := MyPrinter.GetCurrentFont;
```
Store the ID of the current font.

See Also
GetDefaultFont

GetDefaultFont / GetLPP

Declaration

```
function GetDefaultFont : Byte;
```

Purpose

Returns the PrintMode ID of the default font.

Description

The font returned by GetDefaultFont will be enabled when EnableDefaultFont is called. The default font is set by calling SetDefaultFont.

Example

```
if MyPrinter.GetDefaultFont <> MyPrinter.GetCurrentFont then
  MyPrinter.EnableDefaultFont;
```

Enables the default font if it is different than the font currently in use.

See Also

GetCurrentFont SetDefaultFont

Declaration

```
function GetLPP : Dimension;
```

Purpose

Returns the current number of lines per page.

Description

This method returns the current number of lines per page from a data field maintained within the Printer object. It does not request the information directly from the printer. OPPRINT uses the value returned by GetLPP to compute how many CR/LF's are required to reach the next page when the pSoftFormFeed option is enabled.

The number of lines per page may be set by calling SetLPP. Turning on a PrintMode that has the pmChangesLPP option enabled also updates the internal value of lines per page. See AddPrintMode for more information.

Example

```
WriteLn('The current number of lines per page is ', MyPrinter.GetLPP:3:0);
```

Display the number of lines per page currently in use by MyPrinter. Since GetLPP returns a Dimension (either a Real or a Double depending on the $N directive), the value must be formatted not to display characters to the right of the decimal point.

See Also

AddPrintMode SetLPP

Declaration
```
function GetRegisteredType : Word;
```

Purpose
Returns the hardware type registered to this Printer.

Description
The Printer object maintains a data field that is used by routines provided in OPDEVICE to store a hardware type code. The Printer object attaches no special meaning to this field.

Example
```
HardwareType := MyPrinter.GetRegisteredType;
Write('Printer type is: ');
case HardwareType of
  pLaserJet    : WriteLn('LaserJet');
  pDeskJet     : WriteLn('DeskJet');
  pEpsonMX     : WriteLn('Epson MX');
  ...
end;
```
Determines the hardware type registered to MyPrinter and displays it. The constants used within the case statement are interfaced by the OPDEVICE unit.

Declaration
```
procedure GetXRatio : Dimension;
```

Purpose
Returns the X axis scaling ratio.

Description
See the introductory discussion for more information about printer ratios. The default X ratio is 1.0.

Example
```
WriteLn('MyPrinter''s X ratio is ', MyPrinter.GetXRatio);
```
Displays the current X ratio for MyPrinter.

Declaration
```
procedure GetYRatio : Dimension;
```

Purpose
Returns the Y axis scaling ratio.

Description
See the introductory discussion for more information about printer ratios. The default Y ratio is 1.0.

Example
```
WriteLn('MyPrinter''s Y ratio is ', MyPrinter.GetYRatio);
```
Displays the current Y ratio for MyPrinter.

Init

Declaration

```
constructor Init(Name : String; DeviceType : PrnType; OptionFlags : Word);
```

Purpose

Initialize a Printer.

Description

This is the simplest constructor provided by the Printer object. It allocates and links to a BasePrinter object based on the values in Name and DeviceType.

If DeviceType is LPT1, LPT2, or LPT3, Init ignores the Name parameter and allocates a BiosPrinter object on the specified port. If DeviceType is Prn or DiskFile, Init uses the Name parameter to allocate a DosPrinter with the specified name. In this latter case, Name may be 'PRN', 'LPT1', other device names, or a disk file.

OptionFlags specify the Printer's options. See the Constants section above for more information.

Init initializes the hardware type to pPlain. It initializes the X and Y ratios to 1.0. It refers to the following global typed constants (discussed above) to initialize other data fields: DefaultStrikes, DefaultLPI, DefaultCPI, DefaultPoint, and DefaultLPP. It allocates a font stack of size FontStackSize and a BitSet of size pMaxPrintModes.

If the constructor fails, a detailed status code will be found in InitStatus.

Example

```
if not MyPrinter.Init('', LPT1, pBackSpaceAllowed) then begin
  WriteLn('Failed to init printer. Status = ',InitStatus);
  Halt;
end;
```

Initializes MyPrinter to print to LPT1 through a BiosPrinter. The printer supports backspacing.

See Also

InitCustom Load

Declaration

```
constructor InitCustom(Device : BasePrinterPtr; LPI, CPI, Point, LPP : Dimension;
                       OptionFlags : Word);
```

Purpose

Initialize a printer with full control.

Description

Device is the address of a BiosPrinter or DosPrinter object that has been initialized prior to calling Printer.InitCustom. In this way, you can obtain any desired characteristics for the underlying physical layer. LPI, CPI, Point, and LPP specify the default values of lines per inch, characters per inch, character height, and lines per page. Of these, only LPP is used directly by the Printer object when it performs a FormFeed and the pSoftFormFeed option is enabled. All of these values are stored by the object and may be referred to by higher level printing facilities. OptionFlags specify the Printer's options. See the Constants section above for more information.

InitCustom initializes the hardware type to pPlain. It initializes the X and Y ratios to 1.0. It refers to the following global typed constant (discussed above): DefaultStrikes. It allocates a font stack of size FontStackSize and a BitSet of size pMaxPrintModes.

If the constructor fails, a detailed status code will be found in InitStatus.

Example

```
var MyPrinter : Printer;
    BaseDevice : BiosPrinter;
...
BaseDevice.Init(Lpt1);
if not MyPrinter.InitCustom(@BaseDevice,
                            DefaultLPI, DefaultCPI, DefaultPoint, DefaultLPP,
                            0) then begin
  WriteLn('Failed to init printer. Status = ',InitStatus); Halt;
end;
```

This example instantiates MyPrinter. BaseDevice is a static instance of a BiosPrinter, initialized to print to LPT1. Its address is passed to MyPrinter.InitCustom by using the '@' operator. Default values for LPI, CPI, Point, and LPP (interfaced by OPPRINT) are passed to InitCustom. No printer options are specified.

See Also

Init Load

Declaration

```
function InLandscape : Boolean;
```

Purpose

Returns True if in landscape mode.

Description

The internal landscape flag is enabled when a PrintMode with the pmChangesLandscape bit and a non-zero height is turned on. See AddPrintMode for more information.

Example

```
if MyPrinter.InLandscape then WriteLn('Currently in landscape mode')
else WriteLn('Currently in portrait mode');
```

This example displays a message indicating whether MyPrinter is operating in landscape or portrait mode.

See Also

AddPrintMode

LinkToModes / LinkToPrinter

Declaration

```
procedure LinkToModes(var Modes : SingleList);
```

Purpose

Links the object to the specified PrintMode List.

Description

This method replaces the linked list of PrintModes associated with the instance. This is useful in situations where the entire list of available PrintModes needs to change. For example, many laser printers allow sets of fonts to be loaded via font cartridges or by downloading from a computer.

It is the application's responsibility to dispose of the existing list of modes (if desired) before calling LinkToModes.

Example

```
var
  MyPrinter   : Printer;
  LaserModes  : SingleList;
...
  MyPrinter.LinkToModes(LaserModes);
```

Associates PrintMode list LaserModes with MyPrinter. This example assumes that MyPrinter and LaserModes are already instantiated and initialized before the call to LinkToModes.

See Also

DisposeAllModes

Declaration

```
procedure LinkToPrinter(Device : BasePrinterPtr);
```

Purpose

Links the Printer with a new BasePrinter object.

Description

Device is the address of a BiosPrinter or DosPrinter that has been initialized before the call to LinkToPrinter. If the Printer object allocated the current BasePrinter object, it will deallocate it before assigning the new one. Printer will never dispose of a BasePrinter that was attached to it by using LinkToPrinter or InitCustom.

Example

```
var
  MyPrinter     : Printer;
  MyBasePrinter : BiosPrinter;
...
  MyPrinter.LinkToPrinter(@MyBasePrinter);
```

Associates MyBasePrinter with MyPrinter. This example assumes that MyPrinter and MyBasePrinter are already instantiated and initialized before the call to the LinkToPrinter method.

Declaration
```
constructor Load(var S : IdStream);
```

Purpose
Load a Printer from a stream.

Description
S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Printer.Store. Load reads the fixed data of the Printer object, allocates and loads a dynamic instance of a BasePrinter, allocates and loads all PrintModes that were active at the time of the Store, allocates a font stack and an active mode BitSet, reads procedure codes for the printer error handler and other internal routines, and enables the default font, if any was set at the time of the Store.

The stream registration procedure for a Printer is PrinterStream. Any user-written routines associated with the Printer (error handler, positioning primitive, etc.) must be registered before calling Load.

Example
```
uses OpRoot, OpPrnLow, OpPrint;
var  MyPrinter : Printer;
     PS : BufIdStream;
begin
  PS.Init('MYPRN.STM', SOpen, 1024);
  PS.RegisterHier(PrinterStream);
  PS.Get(MyPrinter);
  if PS.GetStatus <> 0 then begin
    WriteLn('Failed to load printer'); Halt;
  end;
  PS.Done;
  ...
end.
```

Instantiates MyPrinter by reading it from the stream PS.

See Also
Store

Declaration
```
function pGetLastError : Word;
```

Purpose
Return the last error code and reset to zero.

Description
This routine should be called after each printer operation if no print error handler is installed. The error codes returned are shown in Appendix D of this manual. See "Printer Error Handling" above for more information about installing a printer error handler.

Example
```
MyPrinter.PrintStr('Send this string to the printer');
ErrorCode := MyPrinter.pGetLastError;
if ErrorCode <> 0 then begin
  WriteLn('Printer error number ', ErrorCode); Halt;
end;
```

pOptionsAreOn / pOptionsOff / pOptionsOn

Declaration

```
function pOptionsAreOn(OptionFlags : Word) : Boolean;
```

Purpose

Return True if all of the specified OptionFlags are on.

Description

This method provides a convenient means of checking to see if specific printer options are enabled.

Example

```
if MyPrinter.pOptionsAreOn(pSupportsHPRules) then begin
  WriteLn('Using HP LaserJet compatible line drawing, boxes and shading');
  ...
end;
```

Displays a message and perhaps performs other actions if MyPrinter is operating in a mode that supports HP LaserJet rules.

Declaration

```
procedure pOptionsOff(OptionFlags : Word);
```

Purpose

Turn off printer options.

Description

One or more option flags can be disabled with this call. There is no harm in turning off an option flag that is already turned off.

Example

```
MyPrinter.pOptionsOff(pPositionable+pSupportsHPRules+pUseDeciPoints);
```

This example disables the positionable, HP rule compatibility, and decipoint option flags for MyPrinter.

Declaration

```
procedure pOptionsOn(OptionFlags : Word);
```

Purpose

Turn on Printer options.

Description

One or more option flags can be enabled with this call. There is no harm in turning on an option flag that is already turned on.

Example

```
MyPrinter.pOptionsOn(pPositionable+pSupportsHPRules+pUseDeciPoints);
```

This example enables the positionable, HP rule compatibility, and decipoint option flags for MyPrinter.

Declaration

```
procedure Position(Row, Col : Dimension; Relative : RelativeType); virtual;
```

Purpose

Position the printhead to the desired place on the page.

Description

If the Printer has the pPositionable option flag set, the position primitive function is called to perform the positioning. (The primitive function generally computes an escape sequence and sends it to the printer.) Otherwise, Position writes an appropriate combination of CR/LF, backspace, and space characters to move the printhead to the specified position.

Relative indicates whether relative (rtCurrent) or absolute (rtAbsolute) positioning is requested. rtCurrent indicates that Row and Col are considered to be offsets from the current printhead position. Row and Col are multiplied by the current values of Y and X ratios, respectively, before the positioning is carried out.

Example

```
uses OpPrnLow, OpPrint;
var  MyPrinter : Printer;
begin
  if not MyPrinter.Init('', LPT1, pBackSpaceAllowed) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus); Halt;
  end;
  MyPrinter.Position(10, 1, rtAbsolute);
  MyPrinter.PrintStr('Hello world');
  MyPrinter.FormFeed;
  MyPrinter.Done;
end.
```

Instantiates a Printer object, positions the printer to row 10, column 1, writes the string 'Hello world', advances to the top of the next form, and calls MyPrinter's destructor.

See Also

PrintStrPos SetPositionPrim

Declaration

```
function PrinterReady : Boolean;
```

Purpose

Returns True if printer ready.

Description

This routine simply calls the PrnStatus method of the underlying BasePrinter object, and returns True if PrnStatus returns zero. Its ability to determine whether a printer is ready is only as good as the underlying BasePrinter's technique.

Example

```
while not MyPrinter.PrinterReady do
  WriteLn('The printer is not ready.  Please correct the error condition.');
  WriteLn('Press any key (<Esc> to abort).');
  if ReadKey = #27 then Halt;
end;
```

Loops until either the BasePrinter associated with MyPrinter reports it is ready or the user aborts by pressing escape.

See Also

BasePrinter.PrnStatus

PrintStr / PrintStrPos

Declaration

```
procedure PrintStr(S : String); virtual;
```

Purpose

Print a string at the current printhead position.

Description

If double strike is enabled, PrintStr prints each character in S, prints a backspace, and prints the character again, for StrikeCount times. Otherwise PrintStr prints the string just once.

Errors encountered while attempting to write characters are considered recoverable. The PrintErrorFunc will be called in the event of an error. If the PrintErrorFunc returns True indicating the error has been corrected, PrintStr will automatically resend the yet unprinted characters. If the PrintErrorFunc returns False, then the rest of the output is not sent.

Example

```
MyPrinter.Position(10, 1, rtAbsolute);
MyPrinter.PrintStr('Hello world');
```

Prints the string 'Hello world' in row 10 at column 1.

See Also

Position

Declaration

```
procedure PrintStrPos(S : String; Row, Col : Dimension;
                      Relative : RelativeType); virtual;
```

Purpose

Print a string at the specified position.

Description

This method generates appropriate calls to Position and PrintStr.

Example

```
MyPrinter.PrintStrPos('Hello world', 10, 1, rtAbsolute);
```

Prints the string 'Hello world' in row 10 at column 1.

See Also

Position

Declaration
```
procedure ResetPage;
```

Purpose
Reset a Printer instance's page position information.

Description
ResetPage is called automatically by the FormFeed method. You may need to call it if you fill a page (thus reaching the top of a new page) without ever calling FormFeed.

Example
```
var
  MyPrinter : Printer;
  I         : Word;
...
for I := 1 to Trunc(MyPrinter.GetLPP) do
  MyPrinter.PrintStrPos('Hello world', I, I, rtAbsolute);
MyPrinter.ResetPage;
```
Calls GetLPP to return the number of lines per page. 'Hello World' is written on each line. ResetPage informs MyPrinter that the printhead is now sitting at the upper left corner of new page.

See Also
FormFeed

Declaration
```
procedure SetDefaultFont(Attr : Byte);
```

Purpose
Sets the default font for this printer.

Description
This method provides a way to set the PrintMode ID to be used as the default font. The default font is enabled with a call to the EnableDefaultFont method. EnableDefaultFont is generally called within a hardware registration procedure (such as the ones provided in OPDEVICE) to push an initial font value onto the font stack. This step ensures that the default font can be restored in the event that another font is enabled and later disabled.

Example
```
MyPrinter.SetDefaultFont(Italics);
```
Sets the default font for MyPrinter equal to the PrintMode ID Italics.

See Also
EnableDefaultFont GetDefaultFont

SetLPP

Declaration

```
procedure SetLPP(LPP : Dimension);
```

Purpose

Sets the number of lines per page.

Description

The internal setting for the number of lines per page is important only if used with a Printer object with the pSoftFormFeed option flag set. In this case LPP is used to determine how many lines to advance to get to the top of the next page. This routine is called internally whenever a PrintMode with the pmChangesLPP flag is turned on.

Example

```
MyPrinter.SetLPP(60);
```

Sets the number of lines per page for MyPrinter to 60.

See Also

GetLPP

Declaration
```
procedure SetPositionPrim(PosFunc : PrinterPosFunc);
```

Purpose
Set printhead positioning function.

Description
In some cases, like the HP LaserJet, the technique used for positioning the printhead is more complicated (and more flexible) than advancing by some number of characters and lines. For the LaserJet, a printer control sequence must be constructed based on the positioning type (absolute or relative) and coordinate system (row/col or decipoints). The function specified by PosFunc handles this task. Printer methods that position the printhead call PosFunc whenever the pPositionable option is enabled.

This routine is primarily used by the hardware registration routines in OPDEVICE.

Example
```
MyPrinter.SetPositionPrim(LaserPosition);
```
Sets the positioning primitive routine for MyPrinter to LaserPosition.

Declaration
```
procedure SetPrintErrorFunc(PEF : PrintErrorFunc);
```

Purpose
Sets the printer error function.

Description
The printer error function registered by this routine is called each time any error occurs. See the discussion of "Printer Error Handling" for more information.

Declaration
```
procedure SetXRatio(Ratio : Dimension);
```

Purpose
Sets the X axis scaling ratio.

Description
Printer multiplies the Col coordinate passed to Position and PrintStrPos by the X ratio before acting upon it. The default X ratio is 1.0. See the introductory discussion of "The Printer Object" for more information.

Example
```
MyPrinter.SetXRatio(2.0);
```
Sets the X ratio for MyPrinter to 2.0.

See Also
SetYRatio

SetYRatio / Store

Declaration
```
procedure SetYRatio(Ratio : Dimension);
```

Purpose
Sets the Y axis scaling ratio.

Description
Printer multiplies the Row coordinate passed to Position and PrintStrPos by the Y ratio before acting upon it. The default Y ratio is 1.0. See the introductory discussion of "The Printer Object" for more information.

Example
```
MyPrinter.SetYRatio(2.0);
```

See Also
SetXRatio

Declaration
```
procedure Store(var S : IdStream);
```

Purpose
Store a Printer in a stream.

Description
S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Printer to the stream S. It writes the fixed data of the Printer object, calls BasePrinter.Store to write its image, stores all PrintModes that have been added, and writes procedure codes for the printer error handler and other internal routines.

The stream registration procedure for a Printer is PrinterStream. Any user-written routines associated with the Printer (error handler, positioning primitive, etc.) must be registered before calling Load.

Example
```
uses OpRoot, OpPrnLow;
var  MyPrinter : Printer;
     PS : BufIdStream;
begin
  if not MyPrinter.Init('PRN', Prn, 0) then begin
    WriteLn('Failed to init printer. Status = ', InitStatus); Halt;
  end;
  MyPrinter.AddPrintMode('Italics', #27'4', #27'5', 0, 0, Italics, 0);
  MyPrinter.SetDefaultFont(Italics);
  PS.Init('MYPRN.STM', SCreate, 1024);
  PS.RegisterHier(PrinterStream);
  PS.Put(MyPrinter);
  if PS.GetStatus <> 0 then begin
    WriteLn('Failed to store printer'); Halt;
  end;
  PS.Done;
  MyPrinter.Done
end.
```

Writes the MyPrinter instance to the stream.

See Also
Load

Declaration
```
procedure TurnOffAllActiveModes;
```

Purpose
Turn off all the currently active print modes.

Description
The Printer object maintains a BitSet of the PrintModes that are currently enabled. TurnOffAllActiveModes scans this BitSet to turn off all whose bits are set.

This ability to turn off all active PrintModes would be useful in a case where the complete list of available PrintModes was about to be changed. It is advisable to call this method before calling LinkToModes if any PrintModes are active (have been enabled and not yet disabled).

Example
```
var
  MyPrinter    : Printer;
  LaserModes   : SingleList;
...
MyPrinter.TurnOffAllActiveModes;
MyPrinter.LinkToModes(LaserModes);
```

Disables all enabled PrintModes, and links to the list of PrintModes named LaserModes. This example assumes that MyPrinter and LaserModes have already been instantiated and initialized.

See Also
LinkToModes

Declaration
```
procedure TurnOffByID(ID : Byte);
```

Purpose
Turn a PrintMode off by ID.

Description
This method scans the linked list of PrintModes to find the one with the specified ID code. If none is found, TurnOffByID does nothing. Otherwise, it calls a PrintMode method that sends the associated disabling control sequence to the printer, then removes the ID from the BitSet of active modes. For the special ID of DoubleStrike, TurnOffByID ends up calling DisableDoubleStrike.

If the pmFont bit of the indicated PrintMode's options is set, TurnOffByID also pops the font from the font stack and re-enables the font at the top of the stack. See the discussion of "Fonts and the Font Stack" early in this section for more information.

Example
```
MyPrinter.AddPrintMode('Italics', #27'4', #27'5', 0, 0, Italics, 0);
MyPrinter.TurnOnByID(Italics);
...
MyPrinter.TurnOffByID(Italics);
```

Adds a PrintMode with an ID of Italics, enables it and later disables it.

See Also
TurnOnByID

TurnOffByName / TurnOnByID / TurnOnByName

Declaration

```
procedure TurnOffByName(Name : String);
```

Purpose

Turn a print mode off by Name.

Description

Works like TurnOffByID except that it scans the linked list for a matching name string rather than an ID code. Matching is case insensitive. Searching by name is slower than by ID code.

Example

```
const
  ItalicsStr = 'Italics';
...
MyPrinter.AddPrintMode(ItalicsStr, #27'4', #27'5', 0, 0, Italics, 0);
MyPrinter.TurnOnByName(ItalicStr);
...
MyPrinter.TurnOffByName(ItalicStr);
```

Adds a PrintMode with the name represented by ItalicsStr, enables it, and later disables it.

Declaration

```
procedure TurnOnByID(ID : Byte);
```

Purpose

Turn a print mode on by ID.

Description

This method scans the linked list of PrintModes to find the one with the specified ID code. If none is found, TurnOnByID does nothing. Otherwise, it calls a PrintMode method that sends the associated enabling control sequence to the printer, then adds the ID to the BitSet of active modes. For the special ID of DoubleStrike, TurnOnByID ends up calling EnableDoubleStrike.

If the pmFont bit of the indicated PrintMode's options is set, TurnOnByID also pushes the font onto the font stack. See the discussion of "Fonts and the Font Stack" early in this section for more information.

Special action is taken if any of the following PrintMode option flags is set: pmChangesLPI, pmChangesLPP, pmChangesCPI, pmChangesPoint, pmChangesTopMargin, pmChangesLandscape, or pmChangesStrike. See AddPrintMode for more information.

Example

```
MyPrinter.TurnOnByID(Italics);
  ...
MyPrinter.TurnOffByID(Italics);
```

Turns on the PrintMode with an ID of Italics and later turns it off.

Declaration

```
procedure TurnOnByName(Name : String);
```

Purpose

Turn a print mode on by Name

Description

Works like TurnOnByID except that it scans the linked list for a matching name string rather than an ID code. Matching is case insensitive. Searching by name is slower than by ID code.

Example

See the example for TurnOffByName.

D. OPDEVICE - Hardware Specific Printer Support

OPDEVICE adds knowledge of specific printers to OPPRINT's general framework. OPDEVICE doesn't declare any objects. Its primary export is a category of routines called hardware registration procedures. A hardware registration procedure configures an instance of a Printer object for a specific printer. OPDEVICE also contains various primitive routines for printhead positioning and the like, but you probably won't need to refer to them directly.

Here's the example from the start of section C, modified to use a hardware registration procedure to configure MyPrinter:

```
program PrinterExample;
uses
  OpRoot, OpPrnLow, OpPrint, OpDevice;
var
  MyPrinter : Printer;
begin
  if not MyPrinter.Init('PRN', Prn, 0) then begin
    WriteLn('Failed to Init, Status = ', InitStatus);
    Halt;
  end;
  GenericEpsonRegister(@MyPrinter);

  MyPrinter.TurnOnByID(Italics);
  MyPrinter.PrintStr('Hello World');
  MyPrinter.Position(50, 1, rtAbsolute);
  MyPrinter.TurnOffByID(Italics);
  MyPrinter.PrintStr('Goodbye World');
  MyPrinter.FormFeed;
  MyPrinter.Done;
end.
```

There are only two differences: we've added OPDEVICE to the uses list; and we call GenericEpsonRegister after instantiating MyPrinter. GenericEpsonRegister makes a series of calls to AddPrintMode to store the control sequences for italics, bold, and other fonts. It also sets the Printer flags to indicate that a generic Epson is not considered to be positionable (see the definitions near the start of section C) but that it supports backspacing. GenericEpsonRegister customizes the character translation table for MyPrinter so that IBM line drawing characters will be translated to their Epson equivalents. Finally, GenericEpsonRegister stores the hardware code pGenericEpson within MyPrinter so that the type of printer can be identified later.

The registration procedures for some of the other printers perform additional actions. The Hewlett-Packard LaserJet routine is the most sophisticated by far: it also defines a procedure pointer for printhead positioning and initializes default spacings and margins.

OPDEVICE is the kind of unit that we'll never be able to finish. It currently supports 14 common printers. We'll continue to add new printers and enhance the features of the ones that are already supported. We hope that if you write a registration procedure and supporting routines for a new printer you'll send it to us so that we can share it with others. See the READ.ME file for the latest news on printer support.

Because all of the registration routines are identical in form, their use is explained only once, under the procedure heading XxxRegister. The following table gives details concerning specific registration routines. It shows the hardware identification code name, the actual code number, the registration procedure name, the printer models to which the registration applies, and

comments specific to that printer's customization. Unless otherwise stated, the Printer options will indicate that the printer is not randomly positionable but allows backspacing.

Constant	Number	Registration procedure	Supported printers

pLaserJet 1 LJRegister All LaserJets
The LaserJet is a positionable printer and also supports rules, boxes, and shading. The registration procedure sets the appropriate flags and installs various primitive handlers to access these capabilities. Printer information relating to LPI, CPI, LPP, character height, and physical top margin is set to the power up defaults for the LaserJet. A PrintMode for the default font is added and it is enabled. Other PrintModes added: Italics, Bold, Dim, Compressed, Underline, and PrinterReset.
A separate procedure, LJFontGRegister, installs a translation table for the line drawing character set of HP's G cartridge. It may be called only after LJRegister has been used to register the Printer instance as a LaserJet.

pDeskJet 2 HPDeskJetRegister HP DeskJet
PrintModes added: Bold, Compressed, Underline, and PrinterReset.

pEpsonMX 4 EpsonMXRegister Epson MX and compatibles
PrintModes added: Italics, Bold, Compressed, Underline, Unidirectional, DoubleWidth, and PrinterReset. Line drawing characters are translated to the standard Epson set.

pEpsonLQ 6 EpsonLQRegister Epson LQ and compatibles
PrintModes added: Italics, Bold, Compressed, Underline, Unidirectional, DoubleWidth, PrinterReset, and LetterQuality. Line drawing characters are translated to the standard Epson set.

pGenericEpson 7 GenericEpsonRegister Generic Epson compatibles
This routine is useful for printers that are considered generic Epson compatibles. This is the lowest common denominator of Epson compatibility.
PrintModes added: Italics, Bold, Compressed, Underline, and PrinterReset.
Line drawing characters are translated to the standard Epson set.

pDiablo 10 DiabloRegister Diablo and compatible daisywheels
PrintModes added: Doublestrike and Underline. The strike count is left at the default. Line drawing characters are translated to ASCII '|-+'.

pProprinterII 11 ProPrinterIIRegister IBM Proprinter II
PrintModes added: Italics, Bold, Compressed, Underline, and PrinterReset.

pKXP1093 12 PXP1093Register Panasonic KX-P1093
PrintModes added: Bold, Underline, and PrinterReset. Line drawing characters are translated to the standard Epson set.

pToshiba321 13 ToshibaP321Register Toshiba P321SL or P341SL
PrintModes added: Italics, Bold, Compressed, Underline, and PrinterReset.

pStarMicronics 14 StarMicronicsRegister Star Micronics
PrintModes added: Italics, Bold, Compressed, and PrinterReset. Line drawing characters are translated to the standard Epson set.

pNec3510 15 Nec3510Register NEC 3510/3530
PrintModes added: Bold, Underline, and PrinterReset. Line drawing characters are translated to ASCII '|-+'.

```
Constant    Number  Registration procedure  Supported printers
-------------------------------------------------------------------------

pNec8800        16  Nec8800Register         NEC 7710/7730/8810/8830
  PrintModes added: Bold, Underline, and PrinterReset. This differs from
  pNec3510 only in the PrinterReset control sequence used. Line drawing
  characters are translated to ASCII '|-+'.

pProwriter      17  ProwriterRegister       C. Itoh 8510
  PrintModes added: Bold, Compressed, Underline, and PrinterReset. Line drawing
  characters are translated to ASCII '|-+'.

pFacit          18  FacitRegister           Facit 4512
  PrintModes added: Bold, Underline, Compressed, and PrinterReset. Line drawing
  characters are translated to ASCII '|-+'.

pDataProducts . 19  DataProductsRegister    DataProducts DP
  PrintModes added: Bold and PrinterReset. Line drawing characters are
  translated to ASCII '|-+'.
```

These routines don't make all features of these printers available. In fact, we must confess that we don't have all (or even many) of these printers; the control sequences were found in other sources and entered without testing on our part.

Printer Libraries

If you're writing an application that makes substantial use of the printer, you'll probably want to support as many devices as you can. Rather than linking in all of the registration procedures, you may find it more efficient to maintain a library of printer objects. As we've mentioned previously, instances of Printer objects can be stored in a stream. Object Professional can combine multiple streams into a random-access library. Putting these two ideas together, you can load preconfigured printers by plucking them out of a library.

Remember that streams are not a panacaea. Although a printer library can hold all of the control sequences, printer mode codes, and character translation tables, it cannot store the fragments of code necessary for hardware-specific printhead positioning. Your program will link these routines from the OPDEVICE unit as usual. The pointer code registration that's necessary for stream management guarantees that the required code is linked in by the compiler.

The following program demonstrates how to write preconfigured Printer objects to a library.

```
program MakeLibrary;  {EXPRNLIB.PAS}
uses
  OpRoot, OpPrnLow, OpPrint, OpDevice;
const
  LibSig = 'PRNLIB';
  LibName = 'PRINTER.OPL';
  MaxEntries = 16;
var
  Lib : Library;
  Dev : Printer;

procedure ConstructorError(ObjectType : String);
begin
  WriteLn('Error creating ', ObjectType, ' Status = ', InitStatus);
  Halt;
end;
```

```
procedure CheckLibError;
var
  Status : Word;
begin
  Status := Lib.GetStatus;
  if Status <> 0 then begin
    WriteLn('Library error. Status = ', Status);
    Halt;
  end;
end;

begin
  {create a new library}
  if not Lib.Create(LibName, 4096, LibSig, MaxEntries) then
    ConstructorError('Library');
  {register object and procedure types in library}
  Lib.RegisterHier(PrinterStream);
  Lib.RegisterHier(LJDeviceStream);

  {write the GenericEpson device to the library}
  if not Dev.Init('', LPT1, 0) then
    ConstructorError('Printer');
  GenericEpsonRegister(@Dev);
  Lib.PutEntry('GENERICEPSON', Dev);
  CheckLibError;

  {write the LaserJet device to the library}
  Deregister(@Dev);
  LJRegister(@Dev);
  Lib.PutEntry('LASERJET', Dev);
  CheckLibError;

  {...write other devices...}

  {close the library}
  Lib.Done;
end.
```

You can easily use this program as the basis for creating your own printer libraries. In fact, the version of MakeLibrary in EXAMPLES.LZH writes all available device types to the library.

MakeLibrary starts by creating a new library file, in this case named 'PRINTER.OPL'. The library will have directory space for up to 16 printers. We give the library a special identifying code of 'PRNLIB' so that other programs like MAKEMENU and MAKESCRN won't be able to open it.

Next, MakeLibrary registers the object types required for Printer objects. Because we intend to create a library entry for the LaserJet, we also use its special stream registration procedure, LJDeviceStream, which registers the required procedure pointers. This wouldn't be necessary if we weren't dealing with the LaserJet device. At this time no other printers supported by OPDEVICE require a separate registration routine like LJDeviceStream.

The first printer type we write to the library is the generic Epson type. To do so, we create an instance of a Printer object and then call the GenericEpsonRegister procedure to set up the appropriate options, control sequences, and character translation. PutEntry writes the customized Printer object to the library. We'll refer to this entry as 'GENERICEPSON' when we access it later. Note that we've initialized this Printer object to write to the LPT1 port using BIOS printing services. The choice of ports isn't critical now, since we can change it after we re-read the object later. After writing the generic Epson printer, we deregister the Printer object

to dispose of its dynamic data and prepare it for reregistration as a LaserJet. The demonstrated sequence could continue for all of the printers in OPDEVICE (up to MaxEntries of them). Finally, the library is closed to assure that all internal buffers are flushed and that the DOS directory is updated.

Now, here's how to initialize a Printer object by using the same library.

```
program ReadLibrary;
uses
  OpRoot, OpPrnLow, OpPrint, OpDevice;
const
  LibSig = 'PRNLIB';
  LibName = 'PRINTER.OPL';
var
  Lib : Library;
  Dev : Printer;
  LptPort : BiosPrinter;

procedure ConstructorError(ObjectType : String);
...

procedure CheckLibError;
...

begin
  {open printer library}
  if not Lib.Init(LibName, SOpen, 4096, LibSig) then
    ConstructorError('Library');
  {register object and procedure types in library}
  Lib.RegisterHier(PrinterStream);
  Lib.RegisterHier(LJDeviceStream);

  {read the LaserJet device and close the library}
  Lib.GetEntry('LASERJET', Dev);
  CheckLibError;
  Lib.Done;

  {set Dev for a different printer port}
  if not LptPort.Init(LPT2) then
    ConstructorError('BiosPrinter');
  Dev.LinkToPrinter(@LptPort);

  {...use Dev...}

  Dev.Done;
end.
```

Lib.Init opens the existing library file and then calls RegisterHier to inform the stream manager of the appropriate data types. Again, the reference to LJDeviceStream is needed only if the LaserJet device is to be used. Lib.GetEntry reads the entry whose directory name is 'LASERJET'. The example then shows how to switch to a different BasePrinter object. In this case, we instantiate a BiosPrinter that will communicate with LPT2. The LinkToPrinter method disposes of the previous (LPT1) BiosPrinter and associates Dev with the new object. At this point, Dev is fully initialized and we can use all of the Printer object's capabilities.

Constants

```
pLaserJet          = 1;
pDeskJet           = 2;
pEpsonMX           = 4;
pEpsonLQ           = 6;
pGenericEpson      = 7;
pDiablo            = 10;
pProprinterII      = 11;
pKXP1093           = 12;
pToshiba321        = 13;
pStarMicronics     = 14;
pNec3510           = 15;
pNec8800           = 16;
pProwriter         = 17;
pFacit             = 18;
pDataProducts      = 19;
```

The hardware registration numbers for all supported printers. The registration number is stored in a data field of the Printer object and can be retrieved with the GetRegisteredType method.

Declaration

```
procedure Deregister(P : PrinterPtr);
```

Purpose

Deallocate dynamic data and reset P to a plain printer.

Description

Call this procedure before registering the printer P as a new printer type. It calls Printer.DisposeAllModes, resets the character translation table, clears the font stack, clears the active mode set, and sets the hardware type to pPlain.

Example

```
EpsonMXRegister(MyPrinterPtr);
...
Deregister(MyPrinterPtr);
LJRegister(MyPrinterPtr);
```

Registers MyPrinterPtr as an Epson MX compatible. Later it deregisters MyPrinterPtr and reregisters it as a LaserJet. This example assumes that either the underlying BasePrinter was adjusted to communicate with a different printer port, or some other mechanism has been used to switch MyPrinterPtr's output to the laser printer instead of the Epson.

Declaration

```
function LJCol2Deci(Col : Dimension; P : PrinterPtr) : Dimension;
```

Purpose

Convert a column number to a decipoint.

Description

This function requires that the Printer P be aware of the active font's correct CPI value.

The sample program FORMTEST.PAS uses this function if it detects that the Printer instance has been registered as a LaserJet.

Example

```
var
  Deci  : Dimension;
...
  Deci := LJCol2Deci(10, MyPrinterPtr);
```

Calculates the decipoint coordinate that corresponds to column 10 and assigns the result to Deci. The decipoint value is calculated given the CPI information maintained by MyPrinterPtr.

See Also

LJDeci2Col
LJRow2Deci
Printer.AddPrintMode

LJDeci2Col / LJDeci2Row / LJRow2Deci

Declaration

```
function LJDeci2Col(Decipoints : Dimension; P : PrinterPtr) : Dimension;
```

Purpose

Convert a decipoint to a column number.

Description

This function is the opposite of LJCol2Deci. Like LJCol2Deci, it requires that the CPI value stored by the Printer object be accurate.

Example

```
Column := LJDeci2Col(Decipoint, MyPrinterPtr);
```

Converts Decipoint to a column number and assigns the result to Column.

See Also

LJCol2Deci Printer.AddPrintMode

Declaration

```
function LJDeci2Row(Decipoints : Dimension; P : PrinterPtr) : Dimension;
```

Purpose

Convert a decipoint to a row number.

Description

This function requires that the Printer P be aware of the active font's correct LPI value.

Declaration

```
function LJRow2Deci(Row : Dimension; P : PrinterPtr) : Dimension;
```

Purpose

Convert a row number to a decipoint.

Description

This function requires that the Printer P be aware of the active font's correct LPI value.

See Also

LJCol2Deci

Declaration

```
procedure XxxRegister(P : PrinterPtr);
```

Purpose

Registers P as a specified hardware device type.

Description

This entry describes all of the hardware registration procedures provided by OPDEVICE. P is the address of a Printer object that has already been initialized as a plain printer. XxxRegister customizes the instance to include specific option flags, PrintModes, character translation tables, a default font, and so on. XxxRegister does not affect the underlying BasePrinter, the X or Y ratios, or the user-written printer error handler.

The entries in the following table show the hardware identification code name, the actual code number, the registration procedure name, and the printer models to which the registration applies. A more detailed version of this table appears in the introduction to this section.

Constant	Number	Registration Procedure	Supported Printers
pLaserJet	1	LJRegister	All LaserJets
pDeskJet	2	HPDeskJetRegister	HP DeskJets
pEpsonMX	4	EpsonMXRegister	Epson MX and compatibles
pEpsonLQ	6	EpsonLQRegister	Epson LQ and compatibles
pGenericEpson	7	GenericEpsonRegister	Generic Epson compatibles
pDiablo	10	DiabloRegister	Diablo and compatible daisywheels
pProprinterII	11	ProPrinterIIRegister	IBM Proprinter II
pKXP1093	12	KXP1093Register	Panasonic KX-P1093
pToshiba321	13	ToshibaP321Register	Toshiba P321SL or P341SL
pStarMicronics	14	StarMicronicsRegister	Star Micronics
pNec3510	15	Nec3510Register	NEC 3510/3530
pNec8800	16	Nec8800Register	NEC 7710/7730/8810/8830
pProwriter	17	ProwriterRegister	C. Itoh 8510
pFacit	18	FacitRegister	Facit 4512
pDataProducts	19	DataProductsRegister	DataProducts DP

Example

See the introductory example program.

See Also

Deregister

E. OPFORM: Printed Forms

OPFORM is the highest level printer support unit. Building on top of OPPRNLOW, OPPRINT, and OPDEVICE, it provides the ability to design printed forms in much the same way that OPENTRY supports data entry screens. Forms are built using a hierarchy of objects whose highest level is called a PrintedForm. Here's the relevant hierarchy:

```
[Root]
  [DoubleListNode]
    [AbstractField]
      PrintField
        StringPrintField*
        CharPrintField*
        WordPrintField*
        BooleanPrintField*
        YesNoPrintField*
        LongPrintField*
        IntPrintField*
        BytePrintField*
        ShortPrintField*
        RealPrintField*
        BCDPrintField*
        ExtPrintField*
        DblPrintField*
        SglPrintField*
        CompPrintField*
        DatePrintField*
        TimePrintField*
        LineField*
          ShadedField*

  [SingleListNode]
    FieldListNode*
    PrintBlock*
    PrintPage*
    PrintedForm

  Report
```

The hierarchy is easier to understand than it looks. For one thing, the objects whose names are surrounded by brackets are described in other sections of the manual and are shown here only to provide a frame of reference. Also, the multitude of types derived from PrintField are very similar to one another. Just like the derivatives of EntryField in the OPFIELD unit, the PrintField derivatives are used to convert a particular Pascal data type into a format that can be used by the higher level objects.

With that in mind, let's take a brief tour through the remaining objects in this hierarchy. The PrintField stores the location, dimensions, and print mode (see OPPRINT) of a single item to be printed. Although a PrintField is associated with a particular variable, it does not know the type of that variable. It is an abstract object, and you would never declare an instance of one. It simply defines data fields and methods that can be shared in common by fields of more specific types: StringPrintField, WordPrintField, RealPrintField, and so on. Through the magic of procedure pointers and virtual methods, however, the numerous specific types can all be treated as a single type, a PrintField. It's worth noting that PrintField's direct ancestor, AbstractField, is shared with the screen-oriented fields described under OPFIELD and OPENTRY. As a result,

the entry screen and printed form systems share the identical code (in OPABSFLD) for converting Pascal variables to strings and merging those strings with picture masks.

Looking now at the next branch of the hierarchy, which descends from the SingleListNode, we first see the FieldListNode type. This is a very simple object used internally during the final process of preparing a field list for output to the printer. We won't need to discuss it in detail.

A PrintBlock's primary component is a list of PrintFields. It introduces an intermediate level of organization into a printed form, a level that doesn't exist in an entry screen. Additional data in a PrintBlock includes a base row and column position, and a user-defined block function that is called just prior to printing the block. The best way to see the use of a PrintBlock is by considering an example: Suppose that you wanted to print multiple mailing labels on a single page. To do so, you would create a PrintBlock whose list of PrintFields was positioned at the upper left corner of the page and included pointers to the name, address, and other variables for a single person. Then you would duplicate that PrintBlock (using supplied methods of PrintedForm) at all desired locations on the page. OPFORM would compute the absolute locations of all the PrintFields for you, but it would leave all of the variables pointing to the single set you initially specified. That's where the user-defined block function comes in. Before printing each block, OPFORM will call your routine which, based on information stored within the block, can retrieve the actual mailing label's data from elsewhere (probably from a database) and store it into the single working variable.

The PrintPage organizes the format of a single page of paper. The PrintPage contains a list of PrintBlocks each of which is a list of PrintFields. Other data associated with a PrintPage includes a header and footer (themselves just instances of a PrintField) and a user-defined page function, which is called just prior to printing the page. The page function might perform actions like updating a page number which will be merged into the header, or incrementing another counter that triggers a whole new page's worth of mailing label data.

As you might expect, PrintedForm contains a list of PrintPage objects. Hence, unlike some other systems, Object Professional's forms can go on for an arbitrary number of pages and page formats. PrintedForm is the object that will receive the most attention throughout the rest of this section. PrintedForm provides methods that allocate, access, and manage all of the lower level objects, so with rare exceptions you'll never need to use those objects directly. Consistent with its central role in the OPFORM system, PrintedForm contains a number of other data fields that specify where and how the form will be printed. Here's a summary of the most important ones:

- bit-mapped option flags that apply to the form as a whole
- top, bottom, left, and right margins
- an internal status variable and a user-defined printer error handler
- default options for the fields being added to the form
- a pointer to an OPPRINT Printer object
- a user-defined pre-form function
- four bytes reserved for user data

Of these, the most important one to mention is the pointer to the OPPRINT Printer object. Through this pointer, PrintedForm inherits all of the capabilities of the lower level printer units already discussed–the ability to print to any port or a disk file, flexible error detection, automatic character translation, PrintModes that provide control sequences for a particular printer, and so on.

There's one more object, Report, which manages a list of PrintedForms. Report is actually a very simple object since most of the detail has already been taken care of before it. The division of duties among the objects yields one interesting benefit: Reports can write their output to multiple printers, a capability which may be useful for many business applications.

To help you get started using OPFORM, let's look at a simple example program that demonstrates some of the features of fields, blocks, and forms.

```
{$V-}                          {<- required for OPFORM}
program FormExample;  {EXFORM.PAS}
uses
  Dos, OpInline, OpString, OpRoot, OpDos, OpDate,
  OpPrnLow, OpPrint, OpDevice, OpForm;
var
  PrnPtr : PrinterPtr;         {pointer to the Printer object}
  FrmPtr : PrintedFormPtr;     {pointer to the PrintedForm object}
  Index : Word;                {record counter}
  DataRec :                    {data record}
    record
      Name : String;
      Addr : String;
      DaysLate : Word;
      Balance : Real;
    end;

{$F+}
function GetNextRecord(P : PrintBlockPtr) : Boolean;
begin
  if Index > 10 then
    GetNextRecord := False
  else
    with DataRec do begin
      {Fill the global record var with data}
      Name := 'Name' + Long2Str(Index);
      Addr := 'Address' + Long2Str(Index);
      DaysLate := 120;
      Balance := 12.34;
      GetNextRecord := True;
      Inc(Index);
    end;
end;
{$F-}

procedure BuildForm;
var
  I : Word;
begin
  {Make a printer device and a form}
  PrnPtr := New(PrinterPtr, Init('DUMP.PRN', DiskFile, 0));
  if PrnPtr = nil then begin
    WriteLn('Failed to init Printer');
    Halt;
  end;
  FrmPtr := New(PrintedFormPtr, Init(1, PrnPtr));
  if FrmPtr = nil then begin
    WriteLn('Failed to init Form');
    Halt;
  end;
```

```
    {Define the fields of the form}
    FrmPtr^.NewPage;
    FrmPtr^.SetHeader(Center('FormExample Header', 80), 1, 1, Dim);
    FrmPtr^.NewBlock(2, 1, GetNextRecord);
    FrmPtr^.AddStringField(1, 'Customer Name:', 2, 3, Bold, 16,
                           2, NullPicture, 2, 20, Dim, 20, DataRec.Name);
    FrmPtr^.AddStringField(3, 'Customer Address:', 3, 3, Bold, 16,
                           4, NullPicture, 3, 20, Dim, 20, DataRec.Addr);
    FrmPtr^.AddWordField(5, 'Days Late:', 4, 3, Bold, 16,
                         6, NullPicture, 4, 20, Dim, 5, DataRec.DaysLate);
    FrmPtr^.AddRealField(7, 'Overdue Balance:', 4, 30, Bold, 16,
                         8, '$###,###.##', 4, 48, Dim, 2, 11, DataRec.Balance);
    FrmPtr^.AddBox(10, 1, 1, 65, 5, 0, 0, SingleLineBox, Bold);
    for I := 1 to 9 do
      FrmPtr^.DupCurrentBlock(2+(I*5), 1);
  end;

  begin
    BuildForm;                    {initialize the form}
    Index := 1;                   {start with the first record}
    FrmPtr^.Process;              {print the report}
    Dispose(FrmPtr, Done);        {clean up}
    Dispose(PrnPtr, Done);
  end.
```

The first thing to notice is the compiler directive {$V-} at the start of the program. This directive is required for almost every program or unit written with OPFORM. If you omit it, you can expect to see a compiler error when you try to add a field of type String to a form.

The complete USES list has been written out so that you can see all of the units involved in using OPFORM. The only unit that might not need to be pulled in is OPDATE; you can remove it if you don't care about date and time variables. To prevent OPFORM itself from using it you must comment out the UseDates conditional define in OPDEFINE.INC.

The example program is typical of many OPFORM applications in that each of the fields in the PrintedForm corresponds to a field in a record variable–in this case DataRec. This is not a hard-and-fast requirement, though. A field in a printed form can correspond to any variable. Our examples use record variables only because most real-world applications do.

Let's skip ahead to the BuildForm procedure, which first initializes the Printer object to which output will go. Rather than repeating a lot of the material of the OPPRINT section, we've chosen to write output to a disk file, DUMP.PRN, without using any special escape sequences for print enhancement. Let it suffice to say that the PrnPtr variable can be initialized just like any other Printer object–by calling a hardware registration procedure, by adding PrintModes, and so on.

Next, a dynamic instance of a PrintedForm variable is allocated and initialized. We use the simple Init constructor to assign an ID code of 1 and the already initialized Printer object to this form. The form ID code becomes useful when building Reports from multiple PrintedForms.

Now the really interesting stuff starts. Each form must begin with a call to NewPage, which allocates a PrintPage and links it into the list of pages. SetHeader specifies a header string and its print attribute and position for this new page. The call to SetHeader exemplifies a technique that is used often in the design of the OPFORM unit: There is also a method named SetHeader in the PrintPage object, and it is this method that does most of the work of

allocating space for the header string and storing the other data about it. The PrintedForm object's version of SetHeader simply assures that there is a current page and passes the call on to that object. In this way we can perform a little more error checking and you don't need to hassle with another set of pointers.

Each page must begin with a call to NewBlock. In many cases the block is just a formality; in this example, however, we're going to make good use of the block to repeat a group of fields for an overdue account form. The first two parameters to NewBlock specify the base row (2) and column (1) coordinates of the block. All of the fields in the block will be relative to these coordinates. The third parameter to NewBlock specifies the name of the user-written block function, which OPFORM calls just prior to printing the block. We'll return to this function in a moment.

The next four calls to AddXxxField generate the fields that will appear on the page. We discuss the parameters to these calls in detail below. For now, we'll note that each call includes a "prompt string" for the field (unchanging text, usually a title next to a number); the prompt's position, print attribute, and width; a picture mask for the data portion of the field; the data portion's position, attribute, and width, and a variable that is associated with the field. In this case, the variables are fields in our DataRec record.

We also add a special kind of field, a "box field," with the call to AddBox. This particular box surrounds the other fields with single-width line drawing characters. How this looks on paper depends on the printer, of course, but since we're writing this to a disk file it'll look just fine using the IBM high-bit ASCII characters.

If you're really following along here, you may have a question now: since the top line of the box is above the first print field that we added, does that mean the printer needs to back up when it comes time to print the box? The answer, fortunately, is No. Why not? First, for printers that won't draw vertical lines, OPFORM splits the box field up into a series of individual print fields, one for each row on which the box appears. Second, OPFORM prints the fields in page-position order, regardless of the order in which they were added to the form. This behavior differs from OPENTRY, which requires that fields be added in positional order.

The loop calling DupCurrentBlock demonstrates the true power of the block concept. Here we create nine copies of the just-defined block, each one of them five rows below the previous one. Remember that although each block is comprised of unique PrintField objects, all of the corresponding fields point to the same variables, in this case the record fields of DataRec.

Note that there's no need for a special call to terminate the design of the PrintedForm.

OK, now we're ready to continue with the main block of code. It first calls BuildForm to open the printer and create the form, a pointer to which is stored in the global variable FrmPtr. Next it sets the value of the global variable Index. This variable will indicate which of the overdue accounts we're going to print next. The call to FrmPtr's Process method prints the form, and the two calls to Dispose deallocate all the memory of the PrintedForm and Printer objects and close the output file.

Let's back up now and look at the GetNextRecord function. PrintedForm's Process method calls this routine just prior to printing each block. Process passes the address of the PrintedForm object, which we don't use in this case, and expects to receive True if the function succeeded and False if it didn't. The pre-block function doesn't print anything. Its main responsibility is to assure that the data fields associated with the block contain the correct values for each subsequent appearance of the block. The global variable Index indicates which record we're processing, so we update DataRec appropriately. GetNextRecord is only designed

to return 10 records; it returns False if we were to ask for more than that, which would cause OPFORM not to print the block.

That lengthy explanation probably gives you a good feeling for the kinds of things you can do with OPFORM. With a few peeks at the method descriptions later in the chapter, you could probably design some pretty nifty looking forms right now. There are lots more features, though, and the remainder of this section will show you how they fit together. Here's a quick rundown of what's ahead:

- adding fields to a PrintedForm
- field options
- field conversion routines
- headers and footers
- additional user hooks
- error handling
- PrintedForms and Streams
- how a PrintedForm works

Adding Fields to a PrintedForm

Because there are over 20 methods for adding PrintFields to a PrintedForm, they are discussed here in groups according to their data type, rather than individually. There is relatively little variation in the parameters passed to these methods, and once you've learned how to use one of them it is very easy to learn how to use the others.

With a few exceptions the first 12 parameters to each of the routines for adding fields are the same. The following table lists these parameters, gives each a brief definition, and notes the methods that don't follow the general rule:

Parameter	Meaning	Field Types
pFieldID	Prompt ID	All except directive fields
Prompt	Prompt message	"
pRow, pCol	Prompt coordinates	"
pAttr	Prompt PrintMode ID	"
pWidth	Prompt width	"
Picture	Picture mask	All except text and line fields
fFieldID	Field ID	"
fRow, fCol	Field coordinates	"
fAttr	Field PrintMode ID	"
fWidth	Field width	All except text, real, and line fields
fDPlaces	Decimal places	All reals except Comp

Parameters that start with 'p' are related to the prompt portion of the field, and those that start with 'f' are related to the data portion of the field.

pFieldID and fFieldID are ID numbers (of type Word) associated with the prompt and data portions of the field, respectively. In contrast to OPENTRY, the programmer assigns these numbers instead of the entry system. The numbers need not be unique within a PrintedForm. Instead they represent a convenient way to identify particular PrintFields, perhaps in order to dynamically modify them. The most common approach will be to number the ID codes sequentially; another logical method would be to assign common codes to related groups. Note that the ID number $FFFF is reserved for OPFORM's internal use. See the PrintedForm methods FindField, FindNextField, and ChangeConversion for more information.

Prompt is a fixed string to be printed whenever the varying portion of the field is printed. Prompt may be an empty string, in which case the values of pRow and pCol don't matter. Note that the prompt and data need not appear on the same row of the form. pRow and pCol specify the position of the prompt using absolute page coordinates in units of Dimension. (See the introduction to OPPRINT for a discussion of this data type.)

pAttr is the ID code of a PrintMode that will be activated prior to printing the prompt and deactivated after. The PrintMode ID code is analogous to the video attribute of an entry field; keep this in mind when we use the Attr abbreviation hereafter.

pWidth specifies the desired width of the prompt in characters; the prompt will be padded or truncated to this width before printing.

Picture is the picture mask for the field's data portion, which describes the format of the field and how it is converted from the Pascal data type to a string. For all simple fields, the picture mask is optional and an empty string may be specified in its position. When no picture mask is specified, fWidth and fDPlaces (for reals) determine how the string is formatted. For further discussion of picture masks, see the OPABSFLD chapter. Although picture masks obviously do not play a role in validating input for the OPFORM unit, they are used just as OPENTRY does for formatting output.

fFieldID, fRow, fCol, fAttr, and fWidth play the same role for the data portion of the field as the previously described parameters do for the prompt.

fDPlaces is used for floating point data fields (Real, BCD, Single, Double, and Extended). If the picture mask for the field has no fixed decimal point, fDPlaces indicates the number of decimal places to be displayed in the output. If the picture mask does contain a decimal point, fDPlaces is ignored.

The last parameter to an AddField method (PrintXxx) is almost always the name of the variable associated with the field. The only exceptions are AddTextField, AddBox, AddDirective, AddLine, and AddShaded, none of which are associated with any variable.

The following subsections list the AddXxxField methods in related groups, show the declaration for each one, and provide additional information where required.

Text Field

The following method adds a simple text field, which is a prompt that has no user variable associated with it.

```
procedure AddTextField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte);
  {-Add a text string field}
```

Simple Fields

The following methods relate to fields that use basic Turbo Pascal types.

```
procedure AddBooleanField(pFieldID : Word; Prompt : String;
                          pRow, pCol : Dimension; pAttr, pWidth : Byte;
                          fFieldID : Word; Picture : String;
                          fRow, fCol : Dimension; fAttr, fWidth : Byte;
                          var PrintBool : Boolean);
  {-Add a field of type Boolean}
```

```
procedure AddByteField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintByte : Byte);
  {-Add a field of type Byte}
procedure AddCharField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintChar : Char);
  {-Add a field of type Char}
procedure AddIntField(pFieldID : Word; Prompt : String;
                      pRow, pCol : Dimension; pAttr, pWidth : Byte;
                      fFieldID : Word; Picture : String;
                      fRow, fCol : Dimension; fAttr, fWidth : Byte;
                      var PrintInt : Integer);
  {-Add a field of type Integer}
procedure AddLongField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintLong : LongInt);
  {-Add a field of type LongInt}
procedure AddRealField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fDPlaces, fWidth : Byte;
                       var PrintReal : Real);
  {-Add a field of type Real}
procedure AddShortField(pFieldID : Word; Prompt : String;
                        pRow, pCol : Dimension; pAttr, pWidth : Byte;
                        fFieldID : Word; Picture : String;
                        fRow, fCol : Dimension; fAttr, fWidth : Byte;
                        var PrintShort : ShortInt);
  {-Add a field of type ShortInt}
procedure AddStringField(pFieldID : Word; Prompt : String;
                         pRow, pCol : Dimension; pAttr, pWidth : Byte;
                         fFieldID : Word; Picture : String;
                         fRow, fCol : Dimension; fAttr, fWidth : Byte;
                         var PrintSt : String);
  {-Add a field of type String}
procedure AddWordField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintWord : Word);
  {-Add a field of type Word}
procedure AddYesNoField(pFieldID : Word; Prompt : String;
                        pRow, pCol : Dimension; pAttr, pWidth : Byte;
                        fFieldID : Word; Picture : String;
                        fRow, fCol : Dimension; fAttr, fWidth : Byte;
                        var PrintYesNo : Boolean);
  {-Add a YesNo field}
```

```
{$IFDEF UseBCD}
procedure AddBCDField(pFieldID : Word; Prompt : String;
                      pRow, pCol : Dimension; pAttr, pWidth : Byte;
                      fFieldID : Word; Picture : String;
                      fRow, fCol : Dimension; fAttr, fDPlaces, fWidth : Byte;
                      var PrintBCD : BCD);
  {-Add a field of type BCD}
{$ENDIF}
{$IFOPT N+}
procedure AddExtField(pFieldID : Word; Prompt : String;
                      pRow, pCol : Dimension; pAttr, pWidth : Byte;
                      fFieldID : Word; Picture : String;
                      fRow, fCol : Dimension; fAttr, fDPlaces, fWidth : Byte;
                      var PrintExt : Extended);
  {-Add a field of type Extended}
procedure AddDblField(pFieldID : Word; Prompt : String;
                      pRow, pCol : Dimension; pAttr, pWidth : Byte;
                      fFieldID : Word; Picture : String;
                      fRow, fCol : Dimension; fAttr, fDPlaces, fWidth : Byte;
                      var PrintDbl : Double);
  {-Add a field of type Double}
procedure AddSglField(pFieldID : Word; Prompt : String;
                      pRow, pCol : Dimension; pAttr, pWidth : Byte;
                      fFieldID : Word; Picture : String;
                      fRow, fCol : Dimension; fAttr, fDPlaces, fWidth : Byte;
                      var PrintSgl : Single);
  {-Add a field of type Single}
procedure AddCompField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintComp : Comp);
  {-Adds a field of type Comp}
{$ENDIF}
```

Date and Time Fields

The following methods relate to time and date fields. If the picture masks are empty, OPFORM supplies 'mm/dd/yy' and 'hh:mm:ss' as defaults.

```
{$IFDEF UseDates}
procedure AddDateField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintDate : Date);
  {-Add a Date field}
procedure AddTimeField(pFieldID : Word; Prompt : String;
                       pRow, pCol : Dimension; pAttr, pWidth : Byte;
                       fFieldID : Word; Picture : String;
                       fRow, fCol : Dimension; fAttr, fWidth : Byte;
                       var PrintTime : Time);
  {-Add a Time field}
{$ENDIF}
```

Line Fields

The following methods add fields that write specialized output sequences for line drawing. See the discussion that follows the declarations for more information.

```
procedure AddBox(FieldID : Word; Row, Col, Width, Height : Dimension;
                 HLineThickness, VLineThickness : Dimension;
                 BoxChars : BoxCharSet; Attr : Byte);
  {-Add a box field}
procedure AddLine(FieldID : Word; Row, Col, Len, LineThickness : Dimension;
                  Horizontal : Boolean; LineChar : Char; Attr : Byte);
  {-Add a line field}
procedure AddShaded(FieldID : Word; Row, Col, Width, Height : Dimension;
                    Intensity : Byte);
  {-Add a shaded area field}
procedure AddDirective(FieldID : Word; Row, Col : Dimension;
                       On : Boolean; Attr : Byte);
  {-Add a print directive}
```

AddBox causes a four-sided box to be drawn on the output page. How the box is constructed depends on whether the pSupportsHPRules flag has been set in the form's underlying Printer. If the flag is not set, PrintedForm breaks the box up into two horizontal lines (including the corner characters) and a series of vertical line segments, one per row. It uses the BoxChars array of characters to determine which symbols to use for each line segment. When the form is printed, the Process method intersperses the line segments with other print fields so that the printhead never needs to back up. If the pSupportsHPRules flag is set, AddBox ignores the BoxChars parameter and instead uses HLineThickness and VLineThickness to specify the thickness of the horizontal and vertical line segments (in decipoints). Row, Col, Width, and Height specify the upper left corner and size of the box in units of character width and height, unless the Printer's pUseDecipoints or pSupportsHPRules flag is set, in which case they are measured in decipoints. pSupportsHPRules is automatically set when a Printer is registered using OPDEVICE's LJRegister routine.

AddLine works much like AddBox, except that only a single segment is drawn. LineChar specifies the character to draw when pSupportsHPRules is not set; LineThickness specifies the thickness (in decipoints) otherwise. Horizontal is True for a horizontal line segment, False otherwise. Row and Col specify the left or upper corner and Len the segment length.

AddShaded adds a shaded box to the output. This feature is ignored unless the Printer's pSupportsHPRules flag is set, since it counts on having the LaserJet-specific escape sequences for such shading. Intensity specifies the shading density, ranging from 1 (barely visible shading) to 99 (almost solid black).

AddDirective adds a special kind of field called a directive, which has neither a Prompt string nor a data variable associated with it. Its sole purpose is to send the enabling control sequence (if On is True) or the disabling one (if On is False) for the specified PrintMode ID code (Attr) when the printhead passes the specified Row, Col coordinate position during the output of a page. If a directive and a field start at the same coordinate, the directive is always printed first.

Field Options

Each PrintField stores a bit-mapped LongInt of option flags. These flags are determined when the field is added (see PromptOptionsOn, FieldOptionsOn, PromptOptionsOff, and FieldOptionsOff in the method reference) or they may be modified later (see FindField, PrintField.pfOptionsOn, and PrintField.pfOptionsOff). Note that when you call most AddXxxField methods, you are actually adding two PrintFields--one for the prompt and another for the data value--and hence you can modify each option independently. We'll introduce the most important options in this section.

pfSuppressZero
Integer fields (Byte, ShortInt, Integer, Word, LongInt) will not be printed if the value is exactly zero.

pfRightJustify
Field is to be right justified in the field's specified width.

pfInhibitAttrOn
When the field is printed, do not enable the PrintMode associated with the field.

pfInhibitAttrOff
When the field is printed, do not disable the PrintMode associated with the field.

pfInhibitString
When the field is printed, do not write the field's string.

pfInhibitPos
When the field is printed, inhibit the head positioning for the field.

pfHidden
Hide the field, i.e., don't print its PrintMode control sequences or the string.

Field Conversion Routines

Every printable field with a user data type has a conversion routine associated with it. The purpose of the conversion routine is twofold: to determine whether the field should be printed at all, and to convert the field from its native data type to a printable string. Conversion routines for PrintFields are much like similarly named routines in OPFIELD, except that the PrintField routines only need to convert one-way.

OPFORM provides default conversion routines for all field types, so you don't need to worry about them unless your form has special requirements. Here are a few situations that might require custom conversion routines:

- field formatting that cannot be represented using picture mask terminology, e.g., negative reals surrounded by parentheses
- dynamic modification of field formatting, e.g., printing a string of asterisks when a real exceeds a preset limit, or switching from fixed point to scientific notation outside of a certain range
- dynamic blanking, e.g., "magic values" that are not to be printed

Dynamic blanking may also be handled by manipulating a field's pfHidden flag, or in some cases by setting the pfSuppressZero option. However, the conversion routine is the central location for managing these kinds of issues and is the most flexible route to take.

A PrintField conversion routine has the following form:

```
{$F+}
function MyPrintConversionFunc(PFP : PrintFieldPtr) : Boolean;
```

The PFP parameter is a pointer to the field to be converted. The function should return True if the field is to be printed; False otherwise.

To give you a better idea of how these work, let's look at one of them. We'll use the default routine PrintWordConvert as our example, since it demonstrates the basic ideas pretty well.

```
{$F+}
function PrintWordConvert(PFP : PrintFieldPtr) : Boolean;
  {-Conversion routine for Word fields}
var S : string;
begin
  with PFP^ do begin
    S := pfLong2Str(Word(pfVarPtr^), 4);
    MergePicture(S, pfString^);
    pfPadField;
    PrintWordConvert := True;
  end;
end;
{$F-}
```

Since a Word is a numeric type, and the goal is to produce a formatted string, the first task is to convert the Word variable pointed to by pfVarPtr into a string. For this we use pfLong2Str, a static "helper" method declared by PrintField, that takes two parameters: a long integer value and a string width. This routine will call one of two routines in OPSTRING, HexL or Long2Str, based on the picture mask in use. If the picture mask indicates that the field is a hexadecimal value, then the string width tells pfLong2Str how many hex digits should appear in the string (2 for Byte/ShortInt, 4 for Word/Integer, 8 for LongInt). Once the value has been converted to a raw string (S), the MergePicture method inherited from AbstractField is called to merge its contents with that of the picture mask and store the result in pfString^. Finally, pfPadField (another helper method of PrintField) pads the field to the width specified when the field was added, left or right justifying it depending on the state of the pfRightJustify option. The standard conversion routines always return True, since the Process method independently checks the state of the pfHidden flag for each field.

In your own conversion routines, you can call any of the methods referenced in this example. Also note that all of the standard conversion routines are interfaced by OPFORM; you can call them to perform most of the work and then handle special cases on your own. Here's a list of the standard conversion routines:

Field Type	Conversion Routine
BCD	PrintBcdConvert
Boolean	PrintBoolConvert
Byte	PrintByteConvert
Char	PrintCharConvert
Comp	PrintCompConvert
Date	PrintDateConvert
Double	PrintDblConvert
Extended	PrintExtConvert
Integer	PrintIntConvert
LongInt	PrintLongConvert
Real	PrintRealConvert
Single	PrintSglConvert
ShortInt	PrintShortConvert
String	PrintStringConvert
Time	PrintTimeConvert
Word	PrintWordConvert
YesNo	PrintYesNoConvert

While writing a custom conversion routine, you should also remember a few other data fields of the PrintField object:

- pfMaxLen: the maximum length of string that may be stored into pfString^

- pfID: the ID code for the field that's being converted
- pfDPlaces : number of decimal places specified when field was added

Also see the methods documented in the PrintField section below.

To activate a custom conversion routine for one or more fields, you use the PrintedForm's ChangeConversion method:

```
procedure ChangeConversion(FieldID : Word; ConvertFunc : PrintConversionFunc);
```

This method finds all fields whose ID codes match FieldID and replaces their conversion functions with ConvertFunc.

Headers and Footers

Each PrintPage may have a header and a footer associated with it. Headers and footers are just instances of PrintFields, so they have positions, print attributes and other options just like other fields. The difference is that a header is printed before all other fields and a footer is printed after all others.

You can add headers and footers immediately after creating a new page, by calling the following methods:

```
procedure SetHeader(HeaderStr : String; Row, Col : Dimension; HAttr : Byte;
  {-Define a header}
procedure SetFooter(FooterStr : String; Row, Col : Dimension; FAttr : Byte);
  {-Define a footer}
```

These are added much like a text field, with a few exceptions. There is no field ID number for headers and footers (they use the reserved value $FFFF). There is also no Width parameter, which is consistent with the fact that, by default, OPFORM performs no formatting (padding or truncation) of the header strings before printing.

Even though headers aren't formatted by default, they are treated exactly like other PrintFields in the sense that a standard (do-nothing) conversion routine is called before printing them. As a result, you can write custom conversion functions for headers as described in the previous section. The most common reason for a conversion routine that applies to a header is to insert a page number in it. Because headers and footers are not identified by field ID codes like other fields, the PrintedForm object provides special methods for assigning user-written conversion routines to them:

```
procedure SetHeaderFunc(Picture : String; HeadFunc : PrintConversionFunc);
  {-Set the HeaderFunc function pointer}
procedure SetFooterFunc(Picture : String; FootFunc : PrintConversionFunc);
  {-Set the FooterFunc function pointer}
```

The Picture parameter is optional here and is provided for the convenience of the conversion function. SetHeaderFunc and SetFooterFunc merely create a copy of Picture on the heap and store its address in the pfPicture field of the header or footer's PrintField. The conversion function may then refer to this picture mask if it needs it.

See the previous section for more information about writing print field conversion functions.

As we've already mentioned, OPFORM prints the header before any other field is printed, and the footer after. It cannot apply the field sorting algorithm (as described later in "How a PrintedForm Works") to these special fields. This leads to the tacit restriction that, for printers whose printhead is not randomly positionable, the header's page position must be above all

other fields and the footer's must be below. Since this is the way headers and footers usually work, this is not an unreasonable restriction, but it's worth keeping in mind.

Error Handling

The technique used to handle errors in the PrintedForm and Report objects is identical to the error handling techniques used by the OPPRINT unit. Please see the discussion of "Printer Error Handling" in the OPPRINT section for further details.

Any fatal error encountered during the course of processing a PrintedForm will cause the operation to abort. If a user error handler is installed, then it will be called to inform the application of the error condition. If no user error handler is installed, an application needs to call the fGetLastError method after the call to the Process method to determine whether the operation succeeded or failed.

To install a user error handler, call PrintedForm's SetErrorHandler method. This method installs the specified error function for both the PrintedForm and its underlying Printer object.

Here's how to add a simple error handler to the introductory example program.

```
{$F+}
function ErrHan(P : PrinterPtr; ErrorCode : Word;
                Recoverable : Boolean) : Boolean;
begin
  if Recoverable then begin
    WriteLn('A recoverable error has occurred. ErrorCode = ', ErrorCode);
    WriteLn('Please correct problem and press any key (<Esc> to abort)');
    ErrHan := (ReadKey <> #27);
  end else begin
    WriteLn('A serious printer error has occurred. ErrorCode = ', ErrorCode);
    WriteLn('Aborting printout...');
    ErrHan := False;
  end;
end;
{$F-}

begin
  BuildForm;                    {Initialize the form}
  Index := 1;                   {Start with the first record}
  FrmPtr^.SetErrorHandler(ErrHan); {Install error handler <-----}
  FrmPtr^.Process;              {Print the report}
  Dispose(FrmPtr, Done);        {Clean up}
  Dispose(PrnPtr, Done);
end.
```

This error handler is very primitive as far as the screen is concerned. It simply writes a message wherever the cursor happens to be. If OPFORM considers the error to be correctable, the error handler waits for a keypress (the reference to ReadKey requires that OPCRT be added to the USES list), and returns True if the user pressed any key besides <Esc>. If ErrHan returns False, Process will not attempt to print the remainder of the form.

The error handler is installed by calling SetErrorHandler at any time after the PrintedForm object is instantiated. If we were to install it earlier (within the BuildForm procedure), the handler would also capture errors that occurred while the form was being built.

Additional User Hooks

We've already described the field conversion routine, which allows you to write routines that gain control as each field is printed. OPFORM also calls optional user-written routines at each

additional level of the form hierarchy: block, page, and form. We've already shown an example of a user-written block function in the introductory example for this section. The block function will be the most commonly used of these hooks.

The page function is called just prior to printing each page of a form. You might use this opportunity to prompt the operator to insert a new sheet into the printer, or perhaps even to prompt for loading an envelope into a laser printer's manual feed bin.

The form function is called once at the beginning and once at the end of each form. This won't be useful unless you've developed a multi-form Report object, since it would be just as easy to call your routines directly. If you *are* using a multi-form Report, the form function may be used to prompt an operator to load a new form type or switch a cable router to another device.

You activate these hook routines by calling the following methods of the PrintedForm:

```
procedure NewBlock(BaseRow, BaseCol : Dimension; BlockFunc : UserBlockFunc);
  {-Starts a new block}
procedure SetPrePageFunc(ppFunc : UserPageFunc);
  {-Specify a function that gets called prior to printing the current page}
procedure SetPrePostFormFunc(ppFunc : UserFormFunc);
  {-Specify a function that gets called before and after printing the form}
```

A UserBlockFunc takes the following form:

```
{$F+}
function MyBlockFunc(PB : PrintBlockPtr) : Boolean;
```

As we've discussed earlier, the block function's usual job is to update a data record that is shared among multiple blocks. After this occurs, the conversion routine of each print field in the block will format its shared user variable and transfer it into a unique string. Note that the formatted strings are never shared between blocks; every field in the form has its own string.

The block function has further control over the printing process through its function result. If the block function returns False, no fields in that block will be printed. This feature is useful if, for example, you've initialized a page to hold 60 one-line, multi-field blocks, but you run out of data after printing only 30 of them. OPFORM will continue to call the block function for another 30 times, but as long as the block function continues to return False, it will not print any output for any of the block fields. However, if the page has remaining fields that are not associated with the block function, such as a footer, it will still print those fields.

A UserPageFunc takes the following form:

```
{$F+}
function MyUserPageFunc(PP : PrintPagePtr) : Boolean;
```

PP is the address of the PrintPage object that is about to be printed. Since the PrintPage object is not documented in this manual, you'll need to look at OPFORM.PAS for more information about it. However, for most of the uses of the page function we've already mentioned, you won't even need to refer to PP; instead you can use a global variable to keep track of a page counter. If the page function returns False, the page is not printed.

Finally, a UserFormFunc is declared as follows:

```
{$F+}
function MyUserFormFunc(PreForm : Boolean; F : PrintedFormPtr) : Boolean;
```

PreForm is True when the function is being called prior to printing a form, and False when it is called after the form is completed. F is the address of the PrintedForm that is being printed. If the form function returns False when PreForm is True, the form won't be printed.

PrintedForms and Streams

The ability to store a PrintedForm to a stream is a significant feature. It is possible, for example, to create a large, complex PrintedForm once, store it in a stream, and reuse it later in other programs. Applications that use many PrintedForms can save code space if the forms are all stored in a Library and loaded from disk when needed.

The only bad news is that it takes a bit more effort to store a PrintedForm in a stream and reload it than it does for most other objects. There are two reasons for the added complexity: the registration of all the PrintFields in the field list, and the registration of the variables associated with the fields. We'll tackle the field list problem first.

You might wonder why the field list poses a problem. Couldn't the registration routine for the PrintedForm (PrintedFormStream) simply register all the field types for you automatically? Yes, it could, but if it did so the compiler would think that every single field type was used in your program–even if only a handful were actually used–and consequently a tremendous amount of dead code would be linked into the program unnecessarily. The convenience of having the PrintedForm register all the field types for you, rather than manually registering only the types you actually use, would come at too high a price for most people. So you have to do it manually.

Each field type that you actually use must be registered before you can store your PrintedForm in a stream. And those same field types must also be registered before you can reload it from the stream. We've done two things to simplify this task. First, we've put together a table (below) that shows the correspondences between the various AddXxxField methods and the stream registration routines that go with them. This table makes it fairly easy to come up with a list of registration routines to call. The second thing we've done is to provide a few routines that register multiple field types at once:

```
AllBasicFieldsStream
All8087FieldsStream
AllDateFieldsStream
AllLineFieldsStream
AllPrintFieldsStream
```

AllPrintFieldsStream registers every field type there is. If you wish, you can just call this one routine and be done with it. But unless you really need all of the different field types (which is highly unlikely) it's unwise to do so.

The other four routines register only a subset of the complete group of field types. The field types registered by these routines are indicated in the table below. Field types with an 'x' in the Basic column are registered by AllBasicFieldsStream. An 'x' in the 8087 column indicates that the field is registered by All8087FieldsStream. An 'x' in the Line column indicates a field is registered by AllLineFieldsStream. And an 'x' in the Date column indicates a field is registered by AllDateFieldsStream.

AddXxxField routine	Stream registration	Basic	8087	Date	Line
AddTextField	none required				
AddBooleanField	BooleanPrintFieldStream	x			
AddByteField	BytePrintFieldStream	x			
AddCharField	CharPrintFieldStream	x			
AddIntField	IntPrintFieldStream	x			
AddLongField	LongPrintFieldStream	x			
AddRealField	RealPrintFieldStream	x			
AddShortField	ShortPrintFieldStream	x			
AddStringField	StringPrintFieldStream	x			
AddWordField	WordPrintFieldStream	x			
AddYesNoField	YesNoPrintFieldStream	x			
AddBCDField	BcdPrintFieldStream				
AddExtField	ExtPrintFieldStream		x		
AddDblField	DblPrintFieldStream		x		
AddSglField	SglPrintFieldStream		x		
AddCompField	CompPrintFieldStream		x		
AddDateField	DatePrintFieldStream			x	
AddTimeField	TimePrintFieldStream			x	
AddBox	LineFieldStream				x
AddLine	LineFieldStream				x
AddShaded	ShadedFieldStream				x
AddDirective	none required				

For example, if you wanted to load or store a PrintedForm that used text, character, and string fields, as well as several 8087 types, you'd make the following calls to initialize the stream manager:

```
var
  S : BufIdStream;
...
S.Init('MYSTREAM', ...);
S.RegisterHier(FormStream);
S.RegisterHier(CharPrintFieldStream);
S.RegisterHier(StringPrintFieldStream);
S.RegisterHier(All8087FieldsStream);
```

The first call to RegisterHier registers the types that PrintedForm itself depends upon and the remaining calls register the field types we care about.

OPFORM interfaces one other stream registration procedure--ReportStream--that you'll care about only if you're storing Report objects to a stream. ReportStream registers everything that FormStream does and the Report object as well.

The second problem involves the registration of all the variables that correspond to the fields in the PrintedForm. In the worst case, if no other solution were provided, you would have to register the address of each and every one of them using RegisterPointer. And you'd have to do it twice: once when the PrintedForm was stored, and once when it was loaded. Although it is not possible to automate this task completely, the PrintedForm is able to simplify it for you using the concept of the "user record." The assumption behind the user record concept is that, in the typical printed form, each PrintField corresponds to a field in a single record variable. If this is so, then the matter of registering the variables can be made much simpler.

PrintedForm supplies a method named SetUserRecord that implements this concept. SetUserRecord stores the base address of a specified user record in an internal data field so that PrintedForm's Load and Store methods can later use it for implicit registration. When a

given field is written to the stream, the Store method will see that the variable is part of the user record and simply save its offset relative to the start of the record.

There is one more twist to consider before using SetUserRecord, however, and that concerns blocks. We've spoken already about the concept of blocks in a form. The duplicated copies of a block share the same user record. As a result, SetUserRecord must work at the block level. Because it does, you can't call SetUserRecord until a block has been created. Each form has at least one block, but the block doesn't exist until you've called NewBlock. Hence, the call to SetUserRecord should be placed shortly after the call to NewBlock.

To make this clearer, let's show how to store the form of the introductory example to a stream. All we really need to look at is the BuildForm procedure:

```
procedure BuildForm;
var
  I : Word;
  FS : BufIdStream;
begin
  {Make a printer device and a form}
  PrnPtr := New(PrinterPtr, Init('DUMP.PRN', DiskFile, 0));
  if PrnPtr = nil then begin
    WriteLn('Failed to init Printer');
    Halt;
  end;
  FrmPtr := New(PrintedFormPtr, Init(1, PrnPtr));
  if FrmPtr = nil then begin
    WriteLn('Failed to init Form');
    Halt;
  end;

  {Define the fields of the form}
  FrmPtr^.NewPage;
  FrmPtr^.SetHeader(Center('FormExample Header', 80), 1, 1, Dim);
  FrmPtr^.NewBlock(2, 1, GetNextRecord);
  FrmPtr^.SetUserRecord(DataRec, SizeOf(DataRec));        { <---------- }
  FrmPtr^.AddStringField(1, 'Customer Name:', 2, 3, Bold, 16,
                         2, NullPicture, 2, 20, Dim, 20, DataRec.Name);
  FrmPtr^.AddStringField(3, 'Customer Address:', 3, 3, Bold, 16,
                         4, NullPicture, 3, 20, Dim, 20, DataRec.Addr);
  FrmPtr^.AddWordField(5, 'Days Late:', 4, 3, Bold, 16,
                       6, NullPicture, 4, 20, Dim, 5, DataRec.DaysLate);
  FrmPtr^.AddRealField(7, 'Overdue Balance:', 4, 30, Bold, 16,
                       8, '$###,###.##', 4, 48, Dim, 2, 11, DataRec.Balance);
  FrmPtr^.AddBox(10, 1, 1, 65, 5, 0, 0, SingleLineBox, Bold, 0);
  for I := 1 to 9 do
    FrmPtr^.DupCurrentBlock(2+(I*5), 1);

  {Write the form to a stream}
  FS.Init('FORMEX.STM',SCreate, 2048);
  FS.RegisterHier(FormStream);
  FS.RegisterHier(AllPrintFieldStream);
  FS.RegisterPointer(1005, @DataRec);
  FS.RegisterPointer(1006, @GetNextRecord);
  FS.PutPtr(FrmPtr);
  FS.Done;
end;
```

The call to SetUserRecord is indicated by an arrow. Any blocks duplicated from this block inherit the same user record, and are thereby registered automatically with the stream manager.

At the end of BuildForm, we create a new stream file, register the data types (taking the easy way out by using AllPrintFieldsStream), and register the addresses of the user data record (DataRec) and the block function (GetNextRecord). Then we call the steam's PutPtr method, which indirectly calls PrintedForm.Store to write out the form, and close the stream file.

The same concept can be applied to multiple independent blocks. Simply make a call to SetUserRecord after each call to NewBlock and specify the name and size of the associated data record.

When the PrintedForm is reloaded from the stream, all that the Load method needs to know in order to fix up the addresses for all the variables is the address for the new user record. Here you have two choices:

1) If you do not register the address of the user record when you Store the PrintedForm, the Load constructor will try to allocate one for you automatically when you go to reload it. You can then use the GetUserRecord method to return the address of the record variable that was allocated. And you can use GetUserRecordSize to verify that the size of the record variable matches your expectations.
2) Alternatively, you can use RegisterPointer to register the address of your user record at the time that you Store it, as shown in the example above. When you go to reload it, you would call RegisterPointer again to register the address of the new user record. If it is not registered, the Load operation will fail.

What about the case where most of the variables associated with the fields are part of a single record, but not quite all? In this case, you can simply use RegisterPointer to deal with all the exceptions.

Here's another version of the BuildForm example procedure that loads the form from the stream.

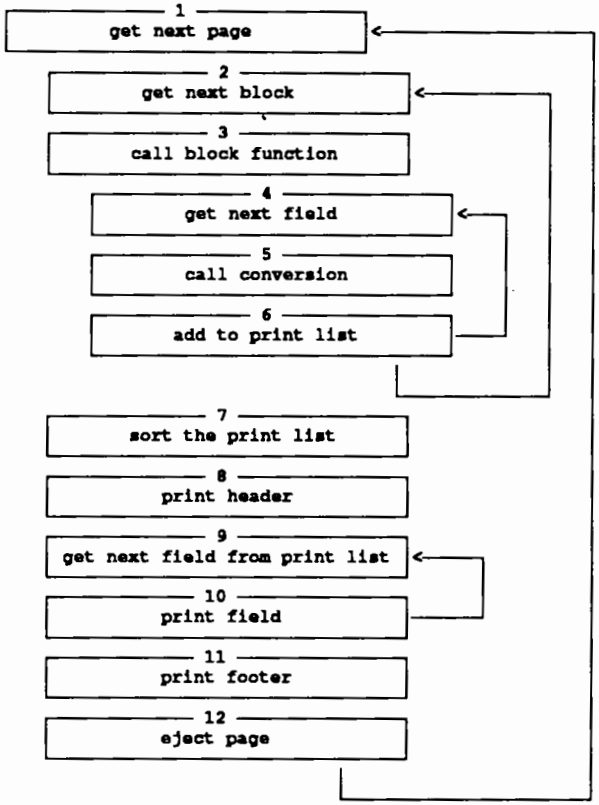
```
procedure BuildForm;
var
  FS : BufIdStream;
begin
  FS.Init('FORMEX.STM', SOpen, 2048);
  FS.RegisterHier(FormStream);
  FS.RegisterHier(AllPrintFieldStream);
  FS.RegisterPointer(1005, @DataRec);
  FS.RegisterPointer(1006, @GetNextRecord);
  FrmPtr := PrintedFormPtr(FS.GetPtr);
  FS.Done;
end.
```

This example takes the approach of alternative 2) above, where we register the address of the user data record before loading the form. As a result, the GetNextRecord function can refer to the global DataRec variable just as before.

How a PrintedForm Works

A diagram illustrating the flow of control while a form is printed may help clarify how OPFORM works. It is our hope that, given an understanding of how control flows from one part of the system to another, you will be better able to understand not only how to use the system, but also how to take advantage of the numerous hooks it makes available.

The flow of control within the PrintedForm.Process method looks something like this:



Printing starts by getting the head pointer of the linked list of pages (box 1). The first block of the page is retrieved (box 2) and the user-defined block function for that block is called (box 3).

If the block function returns True, the processing of the fields begins. The first field is retrieved (box 4) and the conversion routine for that field is called (box 5) to create a printable string from the field's native variable format. If the conversion routine returns True, a pointer to the field is added to a temporary "print list." This list exists only while the page is being printed and its main purpose is to sort the fields into row/column order before writing them to the printer.

Process loops back to get the next field in the block (box 4) and repeats boxes 5 and 6. When all the fields for the current block have been processed, it loops back to box 2 to get the next block. When all the blocks for the page have been processed, it proceeds to box 7, where the list is sorted into positional order. For more information on how the fields are sorted, see the comments in the PrintPage.ppLowerField method (OPFORM.IN1).

After the print list has been sorted, physical printing begins. The first item printed is the header, if one has been defined (box 8). Then, the first field from the print list is returned (box 9), and it is printed (box 10). To print a field, the enabling control sequence of its PrintMode is first sent, then the string, and finally the disabling sequence. Boxes 9 and 10 are repeated until all the fields in the print list have been printed. Then the footer is printed (box 11), if one has been defined.

This completes the page, so it is ejected (box 12) by issuing a form feed. The next page is retrieved from the form (box 1) and the whole process is repeated until there are no more pages.

Although a few details have been omitted, this diagram shows the overall flow of control during printing.

The Report object's Process method repeats this algorithm for each form that is part of the report.

Constants

```
pfSuppressZero   = $00020000; {suppress zeros}
pfRightJustify   = $00010000; {right justify field}
pfInhibitAttrOn  = $00008000; {inhibit calls to turn the PrintMode on}
pfInhibitAttrOff = $00004000; {inhibit calls to turn the PrintMode off}
pfInhibitString  = $00002000; {inhibit the writing of the field's string}
pfInhibitPos     = $00001000; {inhibit the head positioning for a field}
pfHidden         = $00000200; {a hidden field (won't be printed)}
```

These are option flags for objects derived from PrintField. They control the behavior of the field. See the discussion of "Field Options" earlier in this section.

```
DefPadCh : Char = ' ';
```

The character used by default to pad fields.

```
NullPicture : String[1] = '';
```

An empty picture mask.

```
SingleLineBox : BoxCharSet = ('┌└┘─│├');
DoubleLineBox : BoxCharSet = ('╔╚╝═║╠');
```

Box character sets for single and double line boxes.

Types

```
PrintConversionFunc = function (PFP : PrintFieldPtr) : Boolean;
```

The definition of a PrintField conversion function. A print conversion function is called prior to printing each field. See the discussions of "Field Conversion Routines" and "Headers and Footers" for more information.

```
UserBlockFunc = function (PB : PrintBlockPtr) : Boolean;
```

The definition of the user block function. This function is called prior to building the field list for each PrintBlock. See "Additional User Hooks" and "How a PrintedForm Works" above for more information.

```
UserPageFunc = function (PP : PrintPagePtr) : Boolean;
```

The definition of the user page function. This function is called prior to printing each page. See "Additional User Hooks."

```
UserFormFunc = function (PreForm : Boolean; F : FormPtr) : Boolean;
```

The definition of the user form function. This function is called before and after each form is printed. See "Additional User Hooks."

```
PrintFieldProc = procedure (pfPtr : PrintFieldPtr; var D; FPtr : FormPtr);
```

The procedure type definition used by the VisitAllPrintFields method.

```
BoxCharType = frTL..frRR;
BoxCharSet = array[BoxCharType] of Char;
```

The character elements used by the box drawing routines. Note that the constants frTL..frRR are consistent with those defined by OPCRT, but are redeclared in OPFORM to avoid having to use OPCRT in this unit.

```
PrintFieldPtr = ^PrintField;
PrintField =
  object(AbstractField)
    ...
  end;
```

The abstract field type from which all of the actual print fields are derived. There are about 20 such types including StringPrintField, CharPrintField, and so on. See the introductory section "Adding Fields to a PrintedForm" for more information. PrintField is primarily for internal use; however, a few of its methods may be useful to writers of custom conversion routines and other full-featured applications using OPFORM. These methods are documented in a section that follows. Typically you will access a particular PrintField by calling PrintedForm's FindField method to get its address.

```
PrintBlockPtr = ^PrintBlock;
PrintBlock =
  object(SingleListNode)
    ...
  end;
```

An intermediate level object used to organize the fields in a PrintedForm. PrintPage contains a list of PrintBlocks. PrintBlock's methods are used internally and are not documented here.

```
PrintPagePtr = ^PrintPage;
PrintPage =
  object(SingleListNode)
    ...
  end;
```

Another intermediate level object used to organize print fields. PrintedForm contains a list of PrintPages. PrintPage's methods are used internally and are not documented here.

```
PrintedFormPtr = ^PrintedForm;
PrintedForm =
  object(SingleListNode)
    ...
  end;
```

The primary object in OPFORM. Its methods are documented below.

```
ReportPtr = ^Report;
Report =
  object(Root)
    ...
  end;
```

A Report contains a list of PrintedForms. Its methods are documented below.

Convert / GetFieldAttr / GetFieldPos

Declaration

```
function Convert : Boolean; virtual;
```

Purpose

Convert from native type to printable string.

Description

The conversion routine is a virtual method that is called at the time the field is printed. If it returns True, the field is printed; otherwise, the field is ignored. Your program won't call this method directly.

Convert itself does nothing but call a supplied conversion function using a pointer stored within the PrintField. Each object derived from PrintField has a standard conversion function. These functions, as well as the notion of user-written conversion functions, are discussed in the introductory section entitled "Field Conversion Routines."

Declaration

```
function GetFieldAttr : Byte;
```

Purpose

Return the print attribute for this field.

Description

The field attribute is the ID code for a PrintMode that is associated with the Printer object used by the form. The field attribute thus indirectly determines what control sequences will precede and follow the printing of the field.

Declaration

```
procedure GetFieldPos(var Row, Col : Dimension);
```

Purpose

Return the row and column for this field.

Description

GetFieldPos returns the absolute page position that will be used when this field is printed. The coordinates are typically measured in units of character height and width, but they may be reported in decipoints if that option has been enabled.

pfOptionsAreOn / pfOptionsOff / pfOptionsOn

Declaration

```
function pfOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose

Determine whether multiple options are enabled.

Description

This method can be used to test the state of one or more print field option flags. It will return True if all options passed in OptionFlags are enabled.

Example

```
if PfPtr^.pfOptionsAreOn(pfSuppressZero) then
  WriteLn('This PrintField will not print if its data value is 0');
```

Displays a message if the PrintField has the "suppress zeros" option.

Declaration

```
procedure pfOptionsOff(OptionFlags : LongInt);
```

Purpose

Deactivate multiple options.

Description

This method will turn off all options passed in OptionFlags, which may contain one or more options.

Example

```
PfPtr^.pfOptionsOff(pfRightJustify);
```

Clears the pfRightJustify option flag, so that the field will be left justified for all subsequent operations.

Declaration

```
procedure pfOptionsOn(OptionFlags : LongInt);
```

Purpose

Activate multiple options.

Description

This method will turn on all options passed in OptionFlags, which may contain one or more options.

Examples

```
PfPtr^.pfOptionsOn(pfInhibitAttrOn+pfInhibitAttrOff);
```

Options are set to inhibit both the enabling and disabling control sequences for the specified print field. Hence, it will be printed in whatever font is active at the time rather than using the print mode specified when the field was added.

```
PfPtr^.pfOptionsOn(pfHidden);
```

The field will be hidden (it won't be printed at all).

pfSetPicture / Print

Declaration

```
procedure pfSetPicture(NewPicture : String);
```

Purpose

Sets the picture mask.

Description

This method allows the picture mask to be changed "on the fly."

Example

```
PfPtr^.pfSetPicture('######');
```

Changes the picture mask for PrintField pointed to by PfPtr to '######'.

Declaration

```
procedure Print(PrinterP : PrinterPtr); virtual;
```

Purpose

Print the field.

Description

PrintedForm and other objects call this method to print each field. Your application won't call it directly. It's documented here just to describe what happens. Printing a field consists of the following steps:

1. Turn on the field's printer attribute.
2. Position the print head to the field's row and column.
3. Print the field.
4. Turn off the field's printer attribute.

Any of these steps may be disabled. The PrintField option pfInhibitAttrOn disables step 1, pfInhibitPos disables step 2, pfInhibitString disables step 3, and pfInhibitAttrOff disables step 4.

Print handles some cases, such as LaserJet line-drawing, differently.

See Also

Printer.PrintStr

Declaration

```
procedure SetConversion(ConvertFunc : PrintConversionFunc);
```

Purpose

Changes the conversion function pointer maintained by the PrintField.

Description

This allows the conversion function to be changed on the fly. See "Field Conversion Routines" above for more information.

Example

```
{$F+}
function MySpecialConvert(PFP : PrintFieldPtr) : Boolean;
begin
...
end;
{$F-}
...
PfPtr := MyForm.FindField(22, PP, BP);
if PfPtr <> nil then
  PfPtr^.SetConversion(MySpecialConvert);
```

Finds field 22 in MyForm and changes its conversion routine to MySpecialConvert.

See Also

PrintedForm.ChangeConversion Convert

Declaration

```
procedure SetFieldPos(Row, Col : Dimension);
```

Purpose

Sets the row and column coordinates for this field.

Description

This method allows the row and column coordinates for a field to be changed on the fly. You can change the coordinates after a Form has been created, within a user block function, and within a user page function. OPFORM does not rely on the coordinates until it sorts the fields into position order (see "How a PrintedForm Works" above).

Example

```
PfPtr^.SetFieldPos(10, 1);
```

Change the row and column coordinates for the PrintField pointed to by PfPtr.

ChangeConversion / Done

Declaration

```
procedure ChangeConversion(FieldID : Word; ConvertFunc : PrintConversionFunc);
```

Purpose

Change the conversion function for fields with ID number FieldID.

Description

When each print field is added, it is given an ID number. This number need not be unique for each field but usually is. If the FieldID parameter passed to ChangeConversion is an ID used by more than one field, the conversion function for all matching fields will be changed. If an invalid FieldID is passed, this procedure does nothing.

See the discussion of "Field Conversion Routines" for more information.

Example

```
MyForm.ChangeConversion(NameID, NewConversionFunc);
```

This example changes the conversion function to NewConversionFunc for all fields with an ID equal to NameID.

See Also

PrintField.Convert
PrintField.SetConversion

Declaration

```
destructor Done; virtual;
```

Purpose

Dispose of a PrintedForm object.

Description

This destructor deallocates all heap memory used by the PrintedForm instance (and all of its subsidiary objects). Note, however, that Done does not close and dispose of the Printer object associated with the form.

Examples

```
MyForm.Done;
MyPrinter.Done;
```

Dispose of the static object MyForm, freeing all heap memory used by it. MyPrinter.Done disposes of the Printer object used by MyForm, assuming that no other output is to be sent to it.

```
var
  MyFormPtr : FormPtr;
...
Dispose(MyFormPtr, Done);
```

Dispose of the dynamic object MyFormPtr, freeing all heap memory used by it.

See Also

Init
InitCustom
Load

Declaration

```
procedure DupCurrentBlock(NewRow, NewCol : Dimension);
```

Purpose

Duplicates the current block at the new block coordinates.

Description

All the fields added since the most recent block was defined are duplicated into a new block. Field coordinates are updated to use NewRow, NewCol as a base. You can add individual fields to a block after duplicating it, if you want. This section's introduction shows one example of using DupCurrentBlock; the FORMTEST demonstration program shows another.

Example

```
var
  Frm : FormPtr;
...
with Frm^ do begin
  NewPage;
  NewBlock(2, 1, UserBlockFunc);
  AddStringField(11, AcctNumPrompt, 1, 1, Italics, 10,
                 12, NullPicture, 1, 18, Dim, 20, AcctNum);
  AddStringField(13, NamePrompt, 2, 1, Italics, 10,
                 14, NullPicture, 2, 18, Dim, 20, Name);
  DupCurrentBlock(6, 1);
  DupCurrentBlock(10, 1);
end;
```

Starts a new page, starts a new block, and adds two string fields. The block is copied to a new block, which is then copied to a third. This creates three blocks, all of them starting in column 1, at rows 2, 6, and 10.

See Also

NewBlock

DupCurrentPage / fGetLastError

Declaration

```
procedure DupCurrentPage;
```

Purpose

Duplicates the current page.

Description

All of the blocks, fields, headers, and footers added since the last call to NewPage are duplicated and copied to a new page. This is especially useful for long repetitive "line item" reports.

Example

```
var
  Frm : FormPtr;
  I : Integer;
...
with Frm^ do begin
  NewPage;
  NewBlock(1, 1, UserBlockFunc);
  AddStringField(11, AcctNumPrompt, 1, 1, Italics, 10,
                 12, NullPicture, 1, 18, Dim, 20, AcctNum);
  AddStringField(13, NamePrompt, 1, 40, Italics, 10,
                 14, NullPicture, 1, 58, Dim, 20, Name);
  for I := 2 to 50 do
    DupCurrentBlock(I, 1);
  for I := 2 to 10 do
    DupCurrentPage;
end;
```

The two string fields are positioned on the same line to form one "line item." DupCurrentBlock adds blocks to create 50 lines total for the page. DupCurrentPage then adds pages to create 10 total pages in the report. With properly designed block and page functions (see "Additional User Hooks"), this form would correctly print anywhere between 1 and 500 line items.

See Also

DupCurrentBlock NewPage

Declaration

```
function fGetLastError : Word;
```

Purpose

Returns the last error and resets to zero.

Description

A function result of zero indicates no error. A non-zero result may be interpreted by referring to Appendix D. After calling this method, the internal error code is reset to zero.

If you have not installed a PrintedForm error handler, then this routine should be called after each call to a PrintedForm method that could generate an error.

Example

```
Frm^.Process;
ErrorCode := Frm^.fGetLastError;
if ErrorCode <> 0 then
  {handle the error};
```

This example calls Frm's Process method, then assigns ErrorCode with the last error code.

See Also

BasePrinter.PrnStatus

FieldOptionsAreOn/FieldOptionsOff/FieldOptionsOn

Declaration

```
function FieldOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose

Return True if specified default field options are turned on.

Description

The PrintedForm object maintains two LongInt values that are used as the option flags when fields are added. One stores the option flags to be used for the Prompt portion of the field, and the other for the data portion of the field. This method affects the flags maintained for the data portion.

Example

```
if MyForm.FieldOptionsAreOn(pfRightJustify) then
  WriteLn('All subsequent fields added will be right justified');
```

Displays a message if right justification is enabled for subsequent fields.

See Also

FieldOptionsOn

Declaration

```
procedure FieldOptionsOff(OptionFlags : LongInt);
```

Purpose

Turn multiple default field options off.

Description

This method affects the default options for the data portion of fields added hereafter.

Example

```
MyForm.FieldOptionsOff(pfRightJustify);
```

Clear the flag indicating that fields should be right justified. All subsequent fields added will be left justified.

See Also

FieldOptionsOn

Declaration

```
procedure FieldOptionsOn(OptionFlags : LongInt);
```

Purpose

Turn multiple default field options on.

Description

This method affects the default options for the data portion of fields added hereafter.

Example

```
MyForm.FieldOptionsOn(pfRightJustify);
```

Set the flag indicating that fields should be right justified. All subsequent fields added will be right justified.

See Also

FieldOptionsAreOn FieldOptionsOff

FindField / FindNextField

Declaration

```
function FindField(FieldID : Word; var Page : PrintPagePtr;
                   var Block : PrintBlockPtr) : PrintFieldPtr;
```

Purpose

Finds the first field with the specified FieldID.

Description

This method is used in conjunction with FindNextField to retrieve one or more fields with a specific FieldID. A pointer to the PrintField of the first matching field is returned in the function result. This pointer may be used to access PrintField methods or data fields in order to modify the field's options, position, picture mask, or conversion function.

If there is no field with the specified FieldID, FindField returns nil.

The pointers to the PrintPage and PrintBlock associated with the field are returned as well.

Example

```
Field := MyForm.FindField(FieldID, Page, Block);
while Field <> nil do begin
  {hide the field}
  Field^.pfOptionsOn(pfHidden);
  {get the next field with same ID, if any}
  Field := MyForm.FindNextField(Field, Page, Block);
end;
```

This code finds all fields with FieldID as an ID number and hides them.

Declaration

```
function FindNextField(Field : PrintFieldPtr; var Page : PrintPagePtr;
                       var Block : PrintBlockPtr) : PrintFieldPtr;
```

Purpose

Finds the next field with the same FieldID as Field.

Description

This function is designed to be used with FindField. It takes a pointer to a PrintField and finds the next PrintField with the same ID. FindNextField returns a pointer to the next matching field, or nil if none. The pointers to the PrintPage and PrintBlock associated with the field are also returned.

Example

See the example for FindField.

Declaration

```
function fPrinterPtr : PrinterPtr;
```

Purpose

Returns the address of the Printer object associated with this form.

Description

The returned pointer allows you to access all of the methods and data of the Printer object. If you created the form by calling Init, you'll probably have a global Printer variable somewhere, and the result of fPrinterPtr should match the address of that variable. If you loaded the form from a stream, it will have dynamically allocated a Printer; in this case, fPrinterPtr is the only way to access that variable.

Examples

```
MyForm^.fPrinterPtr^.GetPrinterInfo(LPI, CPI, Point, LPP);
```

Returns the lines per inch, characters per inch, character point, and lines per page for the Printer instance associated with MyPrinter.

```
if MyForm^.fPrinterPtr^.PrinterReady then
  MyForm^.Process;
```

Checks whether the printer is ready before printing a form.

See Also

Printer.GetBasePrinter

Declaration

```
function GetFooter : String;
```

Purpose

Returns the footer string for the current page.

Description

If no footer string has been specified with a call to SetFooter, this method returns the empty string. Otherwise it returns either the value specified when the footer was added or the string most recently stored by a footer conversion routine.

Example

```
var
  Foot : String;
...
Foot := MyForm.GetFooter;
if Foot = '' then
  WriteLn('No footer string is currently defined')
else
  WriteLn('The current footer string is ', Foot);
```

This example assigns the current footer string for MyForm to Foot. If it is the empty string, a message is displayed indicating that no footer is defined. Otherwise the footer string is displayed.

See Also

SetFooter SetFooterFunc

GetHeader / GetUserData

Declaration

```
function GetHeader : String;
```

Purpose

Returns the header string for the current page.

Description

If no header string has been specified with a call to SetHeader, this method returns the empty string. Otherwise it returns either the value specified when the header was added or the string most recently stored by a header conversion routine.

Example

See the example for GetFooter.

See Also

SetHeader SetHeaderFunc

Declaration

```
function GetUserData : Pointer;
```

Purpose

Returns the address of the user data field.

Description

Each PrintedForm instance reserves four bytes of data that may be used in any way that an application sees fit. A common use will be to pass record or page number information between the application and user block functions, page functions, or header/footer conversion routines.

The internal data area is typed as a pointer so that it can refer to larger areas of shared storage. Use typecasting to convert it to a LongInt if scalar storage of 4 bytes or less is required.

Example

```
{$F+}
function GetNextRecord(P : PrintBlockPtr) : Boolean;
var Index : LongInt;
begin
  Index := LongInt(P^.GetUserData);
  if Index > 10 then
    GetNextRecord := False
  else with DataRec do begin
    {Fill the global record var with data}
    Name := 'Name' + Long2Str(Index);
    Addr := 'Address' + Long2Str(Index);
    DaysLate := 120;
    Balance := 12.34;
    Inc(Index);
    P^.SetUserData(Pointer(Index));
    GetNextRecord := True;
  end;
end;
```

Avoids using a global variable for the GetNextRecord block function of the introductory example. Note how the result of GetUserData is typecast to a LongInt and how SetUserData typecasts Index to a Pointer.

See Also

SetUserData

Declaration
```
function GetUserRecord : Pointer;
```

Purpose
Returns the address of the user record for current block.

Description
This method returns a pointer to the "user record." It is needed only in cases where a PrintedForm has been loaded from a stream and the Load constructor has had to allocate the record. See the discussion of "PrintedForms and Streams" in the introduction to this section.

See Also
Load

Declaration
```
function GetUserRecordSize : Word;
```

Purpose
Return the size of the user record for current block.

Description
This method returns the size of the "user record" allocated when Load reads a form as described by alternative 1) in "PrintedForms and Streams" above.

See Also
Load

Declaration
```
constructor Init(FormID : Byte; PrinterP : PrinterPtr);
```

Purpose
Initialize a form.

Description
This constructor instantiates a PrintedForm. The form is given the specified FormID, and associated with the already-initialized Printer instance specified by PrinterP. All margins are set to zero, thus disabling any effect they might have. See InitCustom for further details.

Example
```
if not Frm.Init(1, @MyPrinter) then begin
  WriteLn('Failed to init PrintedForm.  Status = ', InitStatus);
  Halt;
end;
```

Instantiates a PrintedForm called Frm with an ID of 1. It will use the static Printer instance MyPrinter, which is assumed to be instantiated.

See Also
InitCustom

InitCustom / Load

Declaration

```
constructor InitCustom(FormID : Byte;
                       TopMargin, BottomMargin,
                       LeftMargin, RightMargin : Dimension;
                       PrinterP : PrinterPtr);
```

Purpose

Initialize a PrintedForm with custom options.

Description

This constructor instantiates a PrintedForm using the specified FormID, starting margin values, and Printer instance.

Field position is adjusted relative to the specified TopMargin and LeftMargin values. For example, if TopMargin is 5.0, all row coordinates specified in AddXxxField calls will have 5.0 added to them. BottomMargin and RightMargin are stored within the instance, but at this time they have no effect on the behavior of PrintedForm.

If InitCustom fails, check the global variable InitStatus for an error code.

Example

```
if not Frm.InitCustom(1, TM, BM, LM, RM, MyPrinterPtr) then begin
  WriteLn('Failed to Init PrintedForm, Status = ',InitStatus);
  Halt;
end;
```

Instantiates a PrintedForm called Frm with a Form ID of 1 and the specified margins. Frm will use the Printer instance pointed to by MyPrinterPtr.

See Also

Init

Declaration

```
constructor Load(var S : IdStream);
```

Purpose

Load a PrintedForm from a stream.

Description

S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to PrintedForm.Store. The byte image includes information about all fields, blocks, and pages of the form.

The various field types used by the PrintedForm must be registered before calling Load. If appropriate, the address of the "user record" should be registered, and so should the addresses of any user-written routines associated with the PrintedForm (conversion routines, header/footer conversion routines, block functions, page functions, and error handlers).

The stream registration procedure for a PrintedForm is FormStream.

For further details, see the discussion of "PrintedForms and Streams" in the introduction to this section.

See Also

Store

Declaration

```
procedure NewBlock(BaseRow, BaseCol : Dimension; BlockFunc : UserBlockFunc);
```

Purpose

Starts a new block.

Description

Every page of a form must have at least one block; therefore, NewBlock must be called after calling NewPage but before adding any fields. The coordinates of fields added after calling this method are adjusted relative to both the top and left margins of the form and also the BaseRow and BaseCol dimensions passed to NewBlock. BaseRow and BaseCol may be zero.

The function specified by BlockFunc will be called immediately before the block is printed. A special BlockFunc called NoBlockFunc is interfaced by OPFORM and may be specified to indicate that no action is to occur.

It is acceptable to have only one block on a page. The number of blocks per page is limited only by available memory.

Examples

```
with Frm^ do begin
  NewPage;
  NewBlock(0, 0, NoBlockFunc);
  AddStringField(11, AcctNumPrompt, 1, 1, Italics, 10,
                 12, NullPicture, 1, 18, Dim, 20, AcctNum);
  ...
end;
```

Defines a new page that will have a single block and no user-defined block action.

```
with Frm^ do begin
  NewPage;
  NewBlock(2, 2, GetNextRecord);
  ...
end;
```

Defines a new page with a block relative to row 2, column 2. Before the block is processed, the function GetNextRecord will be called.

See Also

NewPage

NewPage / Process

Declaration

```
procedure NewPage;
```

Purpose

Start a new page.

Description

Every form must have at least one page; therefore, NewPage must be called after instantiating a PrintedForm and before calling NewBlock, which is also required. All fields added after calling NewPage and NewBlock are added to an instance of the lower level object PrintPage.

To start a new page of the form, call NewPage and NewBlock again. When the form is printed, OPFORM will issue a call to Printer.FormFeed to eject the prior page and load a new one.

Do not call NewPage to terminate a sequence of code that builds a form.

Example

```
•MyForm.NewPage;
 MyForm.NewBlock(0, 0, PrepFirstPage);
 MyForm.AddStringField(...);
 MyForm.NewPage;
 MyForm.NewBlock(0, 0, PrepSecondPage);
 MyForm.AddStringField(...);
```

Creates a form of two pages, each with a single block.

See Also

NewBlock

Declaration

```
procedure Process; virtual;
```

Purpose

Print the PrintedForm.

Description

See the section entitled "How a PrintedForm Works" for more details about the operation of this routine.

Since Process writes output to the physical printer device, it's wise to assure that the printer is ready before calling it and to check for errors afterwards. You can, however, install an error handler with SetErrorHandler and let OPFORM inform you when errors occur.

Example

```
Aborting := False;
while not (MyForm.fPrinterPtr^.PrinterIsReady or Aborting) do begin
  WriteLn('Printer not ready, press any key when ready (<Esc> to Abort)');
  Aborting := (ReadKey = #27);
end;
if not Aborting then begin
  MyForm.Process;
  ErrorCode := MyForm.fGetLastError;
  if ErrorCode <> 0 then
    {error handling};
end;
```

Waits for the printer to become ready, prints the form, and then checks for an error.

See Also

Printer.PrinterIsReady

Declaration

```
function PromptOptionsAreOn(OptionFlags : LongInt) : Boolean;
```

Purpose

Return True if specified default prompt options are turned on.

Description

The PrintedForm object maintains two LongInt values that are used as the option flags when fields are added. One stores the option flags to be used for the Prompt portion of the field, and the other for the data portion of the field. This method affects the flags for the Prompt portion.

Example

```
if MyForm.PromptOptionsAreOn(pfRightJustify) then
  WriteLn('Prompts are being right-justified');
```

Writes a message if the prompt portion of fields being added will be right-justified.

See Also

PromptOptionsOn

Declaration

```
procedure PromptOptionsOff(OptionFlags : LongInt);
```

Purpose

Turn multiple default prompt options off.

Description

This method affects the default options for the prompt portion of fields added hereafter.

Example

```
MyForm.PromptOptionsOff(pfHidden);
MyForm.FieldOptionsOff(pfHidden);
...
```

Enables printing within the default prompt and field options of MyForm. All fields added subsequently will be printed.

See Also

PromptOptionsOn

PromptOptionsOn / SetErrorHandler

Declaration

```
procedure PromptOptionsOn(OptionFlags : LongInt);
```

Purpose

Turn multiple default prompt options on.

Description

This method affects the default options for the prompt portion of fields added hereafter.

Example

```
MyForm.PromptOptionsOn(pfHidden);
MyForm.FieldOptionsOn(pfHidden);
...
```

Disables printing within the default prompt and field options of MyForm. No fields added subsequently will be printed unless they are dynamically enabled.

Declaration

```
procedure SetErrorHandler(ErrorFunc : PrintErrorFunc);
```

Purpose

Sets the user error handler.

Description

This method sets the user error handler for both the PrintedForm instance and the underlying Printer instance.

For further details, see the discussion of "Error Handling" in the introduction to this section.

Example

```
{$F+}
function OurErrorHandler(P : PrinterPtr; ErrorCode : Word;
                         Recoverable : Boolean) : Boolean;
begin
  if Recoverable then begin
    WriteLn('A recoverable error has occurred. ErrorCode = ', ErrorCode);
    WriteLn('Please correct problem and press any key (<Esc> to abort)');
    OurErrorHandler := (ReadKey <> #27);
  end else begin
    WriteLn('A serious printer error has occurred. ErrorCode = ', ErrorCode);
    WriteLn('Aborting printout...');
    OurErrorHandler := False;
  end;
end;
{$F-}
...
Frm^.SetErrorHandler(OurErrorHandler);
```

Declares a PrintErrorFunc named OurErrorHandler and installs it for the PrintedForm instance Frm^.

See Also

Printer.SetPrintErrorFunc

Declaration

```
procedure SetFooter(FooterStr : String; Row, Col : Dimension; FAttr : Byte);
```

Purpose

Define a footer for the current page.

Description

FooterStr specifies the string to print, Row and Col its position on the page, and FAttr its printer attribute. If the footer string will be customized by a footer conversion function, assure that the initial string passed as FooterStr is at least as long as the longest customized version of the string.

See the introductory section "Headers and Footers" for more information.

Example

```
MyForm.SetFooter('Some footer', MyForm.fPrinterPtr^.GetLPP-1, 10, Italics);
```

Defines a footer to be printed using the Italics attribute in column 10 of the next to last line of the page.

Declaration

```
procedure SetFooterFunc(Picture : String; FootFunc : PrintConversionFunc);
```

Purpose

Set the footer function pointer for the current page.

Description

The footer function is called after all other fields have been printed and just before the footer itself is printed. The specified function may customize the footer string within the limits of the original specified length of the footer. SetFooterFunc stores Picture in the picture mask field of the footer's PrintField instance where the footer function may refer to it for its own use.

See the introductory section "Headers and Footers" for more information.

Example

```
{$F+}
function FooterConversion(PFP : PrintFieldPtr) : Boolean;
const
  Page : Word = 0;
begin
  FooterConversion := True;
  Inc(Page);
  with PFP^ do
    pfString^ := Long2Str(Page);
end;
{$F-}
...
MyForm.NewPage;
MyForm.SetFooter(CharStr(' ', 10), MyForm.fPrinterPtr^.GetLPP-1, 2, Bold);
MyForm.SetFooterFunc('', FooterConversion);
...
```

Shows a simple footer function that prints a page number in column 2 of the next to the last line of the page. For this example to work, all pages added to a form must install the same FooterConversion routine, either by calling SetFooter and SetFooterFunc explicitly or by calling DupCurrentPage.

SetHeader / SetHeaderFunc / SetPrePageFunc

Declaration

```
procedure SetHeader(HeaderStr : String; Row, Col : Dimension; HAttr : Byte);
```

Purpose

Set up a header for the current page.

Description

Works just like SetFooter, with the exception that the header is the first field to be printed rather than the last.

See Also

SetFooter

Declaration

```
procedure SetHeaderFunc(Picture : String; HeadFunc : PrintConversionFunc);
```

Purpose

Set the header function pointer for the current page.

Description

Works just like SetFooterFunc, with the exception that the header function is called before any fields on the page have been printed.

Example

```
{$F+}
function HeaderConversion(PFP : PrintFieldPtr) : Boolean;
begin
  HeaderConversion := True;
  with PFP^ do
    pfString^ := LeftPad(CurrentTimeAndDateStr, 1, 80);
end;
{$F-}
...
MyForm.NewPage;
MyForm.SetHeader(CharStr(' ', 80), 1, 1, Bold);
MyForm.SetHeaderFunc('', HeaderConversion);
...
```

Shows a simple header function that prints the string returned by the hypothetical function CurrentTimeAndDateStr on the top right corner of the page.

Declaration

```
procedure SetPrePageFunc(ppFunc : UserPageFunc);
```

Purpose

Sets the function that gets called before the current page is printed.

Description

See the introductory discussion of "Additional User Hooks" for more information.

See Also

NewBlock SetPrePostFormFunc

SetPrePostFormFunc / SetUserData / SetUserRecord

Declaration

```
procedure SetPrePostFormFunc(ppFunc : UserFormFunc);
```

Purpose

Sets the function that gets called before and after printing the form.

Description

This procedure is useful only when printing reports composed of multiple forms.

See the introductory discussion of "Additional User Hooks" for more information.

See Also

SetPrePageFunc

Declaration

```
procedure SetUserData(P : Pointer);
```

Purpose

Sets the user pointer field.

Description

Each PrintedForm instance reserves four bytes of data that may be used in any way that an application sees fit. A common use will be to pass record or page number information between the application and user block functions, page functions, or header/footer conversion routines.

The internal data area is typed as a pointer so that it can refer to larger areas of shared storage. Use typecasting to convert it to a LongInt if scalar storage of 4 bytes or less is required.

Example

See the example for GetUserData.

See Also

GetUserData

Declaration

```
procedure SetUserRecord(var UserRec; UserRecSize : Word);
```

Purpose

Set the address and size of the user record for the current block.

Description

SetUserRecord is of interest only when designing forms that will be saved in a stream. See the discussion of "PrintedForms and Streams" for more information and a complete example.

Example

```
MyForm.NewBlock(1, 1, BlockFunction);
MyForm.SetUserRecord(DataRec, SizeOf(DataRec));
```

Sets the user record to DataRec for the current block.

See Also

GetUserRecord GetUserRecordSize

Store / VisitAllPrintFields

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a PrintedForm in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the PrintedForm to the stream S. The image includes information about all fields, blocks, and pages of the form, as well as the underlying Printer instance.

The various field types used by the PrintedForm must be registered before calling Store. If appropriate, the address of the "user record" should be registered, and so should the addresses of any user-written routines associated with the PrintedForm (conversion routines, header/footer conversion routines, block functions, page functions, and error handlers).

To check for errors that may occur during the Store, call the stream's GetStatus function.

The stream registration procedure for a PrintedForm is FormStream.

For further details, see the discussion of "PrintedForms and Streams" in the introduction to this section.

See Also

Load Printer.Store

Declaration

```
procedure VisitAllPrintFields(pfProc : PrintFieldProc; var D);
```

Purpose

Calls pfProc for each PrintField defined in the form.

Description

VisitAllPrintFields provides a convenient way to update the properties of every field in a form. pfProc specifies a procedure that will be called for each field, and the untyped var parameter D specifies user data that will be passed to pfProc.

pfProc must be compiled with the FAR model (using the {$F+} directive), must be global, and must have parameters that match the following prototype:

```
{$F+}
procedure MyPrintFieldProc(pfPtr : PrintFieldPtr; var D;
                           FPtr : PrintedFormPtr);
```

For each call to MyPrintFieldProc, OPFORM will pass the address of the PrintField in pfPtr, the user data in D, and the address of the form as a whole in FPtr.

Example

```
{$F+}
procedure ClearForAllFields(pfPtr : PrintFieldPtr; var D;
                            FPtr : PrintedFormPtr);
begin
  pfPtr^.pfOptionsOff(LongInt(D));
end;
{$F-}
var UnhideOpt : LongInt;
...
UnhideOpt := pfHidden;
MyForm.VisitAllPrintFields(ClearForAllFields, UnhideOpt);
```

Unhides all fields in the form. The procedure ClearForAllFields typecasts the untyped var parameter D to a LongInt and calls the pfOptionsOff method of the PrintField instance.

Declaration
```
procedure AddForm(fPtr : FormPtr);
```

Purpose
Add a PrintedForm to a Report.

Description
fPtr is the address of an already initialized form. AddForm appends this form to the list of forms comprising the Report.

Example
```
var
  Form1, Form2   : PrintedForm;
  MonthEndReport : Report;
...
{instantiate Form1 and Form2}
...
if not MonthEndReport.Init then begin
  WriteLn('Failed to init Report. Status = ', InitStatus);
  Halt;
end;
MonthEndReport.AddForm(@Form1);
MonthEndReport.AddForm(@Form2);
MonthEndReport.Process;
```
Creates a report consisting of two forms, then prints the report.

Declaration
```
destructor Done; virtual;
```

Purpose
Destroy a report.

Description
This destructor calls the Done destructor for each PrintedForm that was added to the Report. Note that the underlying Printer objects used by each form are not destroyed.

Examples
```
MyReport.Done;
```
Destroy the static Report instance MyReport.
```
Dispose(MyReportPtr, Done);
```
Destroy the dynamic Report instance pointed to by MyReportPtr.

See Also
Init Load

FindForm / Init

Declaration

```
function FindForm(FormID : Byte) : FormPtr;
```

Purpose

Return a pointer to the PrintedForm with the specified ID.

Description

This method searches the linked list of forms and returns a pointer to the PrintedForm instance with the specified FormID. If no PrintedForm with FormID is found, FindForm returns nil.

Example

```
TitleFormPtr := MyReport.FindForm(TitleID);
```

Returns a pointer to the form with the ID TitleID and stores the result in TitleFormPtr.

See Also

AddForm

Declaration

```
constructor Init;
```

Purpose

Create a report.

Description

This constructor instantiates a Report. This constructor will fail only if the report is a dynamic instance and the New routine fails.

Example

```
var
  DailyReport : ReportPtr;
begin
  DailyReport := New(ReportPtr, Init);
  if DailyReport = nil then begin
    WriteLn('Failed to Init Report.  Status = ', InitStatus);
    Halt;
  end;
  ...
```

Instantiates a Report instance called DailyReport. Halts with error message if there is insufficient memory to allocate object.

See Also

Done Load

Declaration
```
constructor Load(var S : IdStream);
```

Purpose
Load a Report from a stream.

Description
S is a properly initialized stream object (a stream file opened for reading, for example). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream.

Load reads the next sequence of bytes from the stream S. These bytes must have been written by a previous call to Report.Store.

The various field types used by the underlying PrintedForms must be registered before calling Load. All requirements for registration of these forms must be met.

The stream registration procedure for a Report is ReportStream. ReportStream automatically calls FormStream.

For further details, see the discussion of "PrintedForms and Streams" in the introduction to this section.

Example
```
Stm.Get(SalesReport);
if Stm.GetStatus <> 0 then begin
  WriteLn('Failed to load Report');
  Halt;
end;
```

Attempts to load SalesReport from the stream Stm. Halts with error message on any errors.

See Also
PrintedForm.Load

Declaration
```
procedure Process; virtual;
```

Purpose
Print the report.

Description
Process calls the Process method of each form added to the report. Note that each form calls its own user-defined pre/post-form function, which may be used to prompt for new paper types or printer switching.

Example
```
MyReport.Process;
```

Prints MyReport.

See Also
PrintedForm.Process

rGetLastError / Store

Declaration

```
function rGetLastError : Word;
```

Purpose

Get last error code and clear internal status variable.

Description

If the function result is zero, then no error has occurred. Otherwise, the error code may be interpreted in accordance with Appendix D.

Example

```
MyReport.Process;
ErrorCode := MyReport.rGetLastError;
if ErrorCode <> 0 then
  {error handling};
```

Calls the Process method for MyReport and then checks for an error condition.

See Also

PrintedForm.fGetLastError

Declaration

```
procedure Store(var S : IdStream);
```

Purpose

Store a Report in a stream.

Description

S is a properly initialized stream object (a stream file opened for writing, usually). Note that S can be any descendant of the IdStream type, which includes DosIdStream, BufIdStream, and MemIdStream (although you typically wouldn't write to a MemIdStream).

Store writes an image of the Report to the stream S.

The various field types used by the underlying PrintedForms must be registered before calling Load. All requirements for registration of these forms must be met.

The stream registration procedure for a Report is ReportStream. ReportStream automatically calls FormStream.

For further details, see the discussion of "PrintedForms and Streams" in the introduction to this section.

Example

```
MyReport.Store(Stm);
```

Store the Report instance MyReport in the stream Stm.

See Also

Load PrintedForm.Store

F. Sample Programs

Currently, there is only one sample program in **Object Professional** that illustrates the use of OPFORM and the other units described in this chapter, FORMTEST.

FORMTEST

Unlike many of the other sample programs provided with **Object Professional**, FORMTEST is not a general purpose utility. It is designed solely to illustrate the capabilities of the OPFORM, OPDEVICE, OPPRINT and OPPRNLOW units.

FORMTEST implements a sample report which shows delinquent accounts. The main purpose of the example is to illustrate the steps involved in creating a line item based report.

FORMTEST accepts several command line options, which are summarized in the following table. Further discussion follows.

Option	Purpose
/?	Display the available options
/Ln	Specify the LPT number (where n is 1, 2, or 3)
/Nfilename	Specify the name of the print file or device
/Onxx	Specify the printer ready test to use. n is a test number ranging from 0 to 4. If the test number is 4, then xx is the success mask in hex.
/R	Read the form from a stream
/S	Suppress form feeds
/Tx	Specify the printer hardware type, where x is: L for LaserJet compatible M for Epson MX F for other Epson compatible I for IBM Proprinter D for Diablo compatible
/W	Write the form to a stream

Run FORMTEST with the /? option first to see if any new options have been added since the manual was printed.

/L indicates that BIOS services should be used to print. The LPT number indicates which port will be written to.

/N indicates that DOS services should be used to print. The filename that follows may be PRN, LPT1, LPT2, LPT3, any other valid DOS character device, or an actual filename.

If neither /L nor /N is specified, FORMTEST behaves as if the '/Ndump.prn' option were given. It's incorrect to specify /L and /N at the same time. For example, if you specified /Nlpt2 /L1 you'd be asking to print to two different places at once.

When using the /L option, you can also use the /O option to specify which printer ready test should be used. The OPPRNLOW section discusses the exact meaning of the different test numbers. The test defaults to number 1. If the specified printer test number is 4, then the two digits immediately following the 4 are interpreted as the test's success mask. Here are some examples of valid /O options:

Option	Meaning
/O1	Use printer test 1 (default)
/O490	Use printer test 4, with $90 as a success mask
/O410	Use printer test 4, with $10 as a success mask
/O0	Do not use a printer test, assume it is always ready

The /T option causes the Printer instance used by FORMTEST to be registered using the specified hardware type. Any PrintModes predefined by OPDEVICE for that printer type will be available to the form. The most impressive example occurs using /TL to register the printer as an HP LaserJet compatible. This allows the form to use advanced LaserJet positioning, line drawing, boxes, and shading. If no /T option is specified, FORMTEST defaults to a "plain" printer, which generates no control sequences but does use the IBM line drawing characters for boxes.

The /W and /R options relate to streams. Streams are read and written to a file named FORMTEST.STM. If the /W option is used, the form is written to the stream file after it has been instantiated and initialized. If the /R option is used, then the form is instantiated from the instance stored in FORMTEST.STM. You must run FORMTEST once with /W before you can run it with /R.

The following are sample FORMTEST command lines, with a brief description of each:

```
FORMTEST /L1 /O3 /TD
```

Specifies a Diablo compatible printer attached to LPT1, using printer test 3.

```
FORMTEST /NPRN /W /TI
```

Specifies a Proprinter compatible with output sent to the PRN device. Also writes a stream file for the form.

```
FORMTEST /L2 /O0 /TL
```

Specifies a LaserJet compatible attached to LPT2. No printer test will be used (the printer will always be considered ready).

```
FORMTEST /R
```

Reads the form from a stream and prints it to the file DUMP.PRN.

```
FORMTEST /Nbilling.rep /TL
```

Writes the form to a file named BILLING.REP. The disk file will contain printer control sequences for a LaserJet compatible printer.

To see some of the inner workings of the PrintedForm, Printer, and BasePrinter objects, try a little experiment. First, type FORMTEST by itself with no command line options. Look at the generated file DUMP.PRN (you can use any text file editor or browser). Now run FORMTEST again, passing the command line option /TL to use the LaserJet escape sequences. When you look at DUMP.PRN again, you'll see what a difference just changing the Printer definition can make to the generated output.

Index

0

8087 reals 11-17
8x8 font 3-22, 35
<CtrlBreak> 4-152
{$N-} 2-9
{$V-} 6-112, 6-176

A

Action routines (for selectors) 6-85
Active coordinates (for windows) 4-70
Active window 4-56
Alternate draw 4-75
Array 8-154
Array, page structure 8-166
Array, priorities 8-157
Array, optimized 8-154
Array, performance 8-156
Array, range checking 8-158
ArrayField 6-29
ASCIIZ strings 9-35
Auto-advance 6-22
Auto-execution of commands 3-91

B

Background task hook 3-89
Backward links 6-74
BCD 14-2
BcdField 6-29
BEST 6-249
Binary Coded Decimal 14-2
Bit flags 14-28
Bit sets 8-50
Blinking attribute 3-6, 3-36
Block copy hook (for help) 4-411
Block function (for printer) 7-80
Block-oriented commands 5-1
BooleanField 6-30
Border 4-8, 4-21, 4-50
Box field 7-70
Boyer-Moore algorithm 9-4
BROWSE 5-80
BrowseStatusProc 5-63
Browser commands 5-57
Browsing 5-1
Burns, Philip R. 14-3
Buttons (see Mouse, buttons)
ByteField 6-30

C

C. Itoh 8510 printer 7-59
Calculated fields 6-124
Calculator 1-30, 14-3
Calendar 1-35
CALIBRAT.LZH 7-6
Calibration, printer 7-6
CGA 3-7
CharEditor 6-52, 6-119, 6-181
CharField 6-31
Character pitch 7-27
Character point 7-27
Characters per inch 7-27
CheckBreak 3-6
CheckEOF 3-6
CheckSnow 3-6
ChoiceField 6-31
Circular linked list 8-29
Cloning 14-16
Cluster size 4-292
Code numbers for objects 8-96
Color names 3-49
ColorMap 3-8
ColorSet 2-8, 3-45
ColorSet and streams 4-160
Commands, codes 3-83
Commands, general 4-140
Commands, keyboard handling 2-8
CompField 6-32
Compiling large programs 15-24
CompuServe 1-45, title-page
Conditional defines 2-9, 4-22, 4-57
Configuration data 2-8, 14-16
Constructors 2-5
Context sensitive help 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Conventions (for this manual) 2-4
Conversion routine (for data entry) 6-15, 6-141
Conversion routine (for printer) 7-77
Copy file 10-6
Cross-reference (for help) 4-367
CRT 3-3
Current field (for data entry) 6-59
Current item procedure (for PickList) 4-312, 4-330
Custom string procedure (for PickList) 4-309
Customizing programs 14-16

D

Data entry 1-2, 6-1
Data entry, commands 6-136
Data entry, fields 6-15
Data entry, nested screens 6-127
Data entry, overview 6-140
Data entry, screens 6-110
Data objects 1-2, 8-1
DataProducts DP 7-59
Date variables 6-206
Date, valid ranges 6-207

DateField 6-32
DateStField 6-33
Decipoint 7-28
Derived types 2-4
DESKPOP 1-24
DesqView 3-1, 3-8
Destructors 2-5
Diablo daisywheel printer 7-27, 7-58
DirectVideo 3-6
Directory 10-18
Directory, changing 4-253
Directory, commands 4-243
Directory, entry (for libraries) 8-127
Directory, format 4-278
Directory, list 4-241
Directory, sorting 4-246
Directory, tree format 4-282, 4-293
Directory, window 4-281
Disk transfer address (DTA) 10-1
Disk class 10-13
Disk sectors 10-22, 10-31
Divider bars 4-39
DPINST 1-40
DOS critical error handler 11-16
DOS environment 10-1
DOS services 11-15
DOS services for printer output 7-4
DOS shell 10-27
DOS unit 10-1
DosBusyFlag 11-27
DosCriticalFlag 11-27
DoubleField 6-33
Doubly linked list 8-29
Draw, menu 4-306
Draw, window 4-72

E

EDITOR 5-79
Editor 1-33, 5-19
EGA 3-1, 3-7, 3-22, 3-35
EMS 8-154, 12-1, 12-3
Enhanced keyboards 3-5, 10-2, 10-42
ENTRY 6-246
Entry screens 6-1
Entry screens and streams 6-133
Entry screens, wrap mode 6-62, 6-80
EntryField 6-20
Epson LQ printers 7-58
Epson MX printers 7-58
Epson dot matrix printers 7-27, 7-58
Error categories 8-150
Error codes 4-141, 8-150, 15-10
Error handling 2-7, 4-140, 7-79, 8-150, 8-158
Error handling, default 4-156
Error message strings 8-150
Error types 15-10
Exchanging values 14-32
ExecDos 10-1
Exit command 6-141, 4-152, 4-157
Expanded memory (see EMS)
Exploding windows 4-77, 4-333
Extended memory 12-1
ExtendedField 6-33
Extension (file) 9-11, 9-13, 9-18
External Interfaces (for TSRs) 11-19, 11-41

F

Facit 4512 printer 7-59
Far calls 14-34
FastDispose 2-9
Field, adding 6-114
Field, attributes 6-58
Field, conversion routines 6-40, 7-76
Field, definitions 6-4
Field, delimiters 6-123
Field, derived types 6-29
Field, drawing routines 6-48
Field, editors 6-4, 6-15, 6-52, 6-141
Field, hidden 6-56, 6-58
Field, ID 6-55, 7-96
Field, invisible 6-57
Field, links 6-60
Field, lists 6-55
Field, movement commands 6-60, 142
Field, options (for data entry) 6-21, 6-122, 7-86
Field, protected 6-56, 6-58
Field, selecting 6-1, 6-3, 6-55, 6-58
Field, user-defined 6-130
Field, validation routines 6-43, 6-44
File attribute 4-242
File browser 5-1, 5-56
Filename 9-18
Filename filtering 4-244
Filename formatting 4-246
Flags 14-44
Float 9-2
Font 7-28, 7-32
Font, stack 7-32
Font, changing 7-27
Font8x8 3-6
Footers 7-78
Formatted strings 9-14, 9-21, 9-49
FORMTEST 7-113
FourByteDates 2-9
Frame 4-8, 4-20 (see Window)
Frame-relative writing 4-81, 4-114
Free pool, window numbers 4-129

G

Generating documentation 6-244

Generating menus 4-420
Generating source code 6-244
Global validation routine 6-126, 6-156
GRAPH3 3-3
GUIDE 1-17

H

Hashing 8-68
Header, more (for PickLists) 4-208, 4-384
Header, printer 7-78
Header, spanning 4-39
Header, window 4-21
Heap usage 2-7
Help 1-17, 4-361
Help, bound in 4-372, 4-394
Help, context-sensitive 2-8, 3-90, 4-138, 4-141, 4-312, 4-361, 4-399
Help, cut/paste 1-21, 4-411
Help, file 4-375
Help, file control characters 4-367
Help, file directives 4-364
Help, index 4-138, 4-365
Help, paged 4-406
Help, previous topic stack 4-377, 4-405
Help, scrolling text 4-409
Help, topics 4-298, 4-365
Hexadecimal string 9-16
HP DeskJet 7-58
HP LaserJet 7-27
Hot spot 4-25
Hot spot, for help 4-377
HotCode 4-91
Hotkeys 11-13
HugePtr 12-12

I

ICD Files 2-8
InitAllUnits 2-9
InitStatus 8-151
Inline macros 14-1, 28
Inline procedures 13-1
INSTALL.EXE 1-10
Installation 1-7, 1-10
IntegerField 6-34
International currencies/conventions 6-11
Interrupt service routines 11-1
Interrupt vector 11-1, 11-4
Interrupts and TSRs 1-3
Item key (for menus) 4-298

J

Julian date 6-208
Jump marker 14-28

K

Key assignment 3-87, 3-112
Key codes 10-42
Keyboard macros 1-3, 13-1
Keyboard status flags 3-27
Keystrokes, commands 3-82

L

LHARC 1-5, 14-28
Landscape mode 7-28
Large array (see Array)
LaserJet printers 7-33, 7-58
LastMode 3-6
Leading 7-28
Letter sets 9-8
Library 8-127
Library directory 8-127
Line editor 6-1, 6-176
Linked list 8-29
Literals (see Picture masks)
LoadableColorSet 4-2
LongIntField 6-34
Lotus style menus 4-297

M

Macro, chaining 13-12, 13-21
Macro, editor 13-15
Macro, engine 13-1, 13-8
Macro, file 13-11, 23
Macro, nested 13-1
Macro, playback 13-18
Macro, recording 13-10
Macro, scrap 13-18
Make file (installation) 1-7
MAKEHELP 4-369
MAKEMENU 4-420
MAKESCRN 6-228
Mapping, video attributes 3-28
Mask (see Picture mask)
Mask, characters 6-5
Mask, date 6-6
Mask, directory header 4-251
MCGA 3-7
Memo 5-1, 5-79
Memo field 5-2, 5-6
Memo window 5-22
Memory allocation 1-3, 12-1
Memory resident programs (see TSR)
Menu, building 4-298
Menu, commands 4-304
Menu, customization 4-301
Menu, Lotus style 4-297
Menu, protected items 4-203, 4-308
Menu, systems 4-297
MergePicture 6-14

Methods, static 2-4
Methods, virtual 2-4
Mickeys 3-60
Micronics printer 7-58
MODE 7-5
Modules (installable) 11-19
Mono 3-6
MonoHerc 3-7
Mouse, buttons 3-54, 3-68
Mouse, coordinates 3-55
Mouse, cursor 3-53
Mouse, drivers 3-59
Mouse, droppings 3-56
Mouse, event Handlers 3-57
Mouse, support 2-8, 3-1, 3-88
Mouse, windows 3-55
MultiLineField 6-35, 6-125, 6-128
Multiple choice lists 4-196
Multitasking system 3-8

N

Naming conventions (for this manual) 2-6
NEC 3510/3530 printers 7-58
NEC 7710/7730/8810/8830 printers 7-59
NestedField 6-35
NullValidation 6-44
NumberEditor 6-52, 120, 181
Numeric coprocessor 11-3
Numeric field 6-21

O

Object Professional Library 8-129
Object, code numbers 8-96
Object, hierarchy 15-7
Object, version number 8-96
Object-oriented I/O 8-90
Object-oriented data structures 8-2
Octal string 9-23
OmniView 3-1, 3-8
OPL 8-129
OS 14-28
Overlays 11-17, 15-14

P

Packed image 4-165
Packed window 4-436
Packed window, bound in 4-168
PackedWindow 4-164
Page frame 12-7 (see EMS)
Panasonic KX-P1093 printer 7-58
Pasting help text 1-21
Path list 4-241
Pathname 9-7, 9-16, 9-19
Periscope 11-16
PGC 3-7
PickHorizontal 4-194
PickList 4-186
PickList field 6-128
PickList, orientations 4-194
PickList, user move hook 4-203
PickListFields 2-10
PickSnaking 4-194
PickVertical 4-194
Picture mask (for printing) 7-2
Picture mask 6-4, 141, 241
Picture mask, semi-literal characters 6-5
Picture mask, user defined characters 6-6
Pitch, character 7-27
PlaybackDelay 13-3
Point 7-28
Point, character 7-27
Pointer arithmetic 14-1
PointerStack 8-6
POPHELP 1-17, 11-35, 13-23
Popup, dispatcher 11-13
Popup, handles 11-14
Popup, routines 11-13, 14
Portrait mode 7-28
Post-edit routines 6-41, 6-124, 6-141, 6-150
Post-select routines 6-86
Pre-edit routine 6-113, 6-123, 141
Pre-select Routines 6-86
PRINT.COM 10-1
Printing 1-2, 7-1
Printing, calibration 7-6
Printing, custom hook
Printing, error handling 7-1, 7-34
Printing, fields 7-2
Printing, forms 7-66
Printing, hardware registration 7-57
Printing, lines and boxes 7-27
Printing, modes 7-2
Printing, overview 7-84
Printing, page function 7-80
Printing, printer libraries 7-59
Printing, streams 7-35, 7-81
Printing, user data field 7-98
Printing, user hooks 7-79
Procedure pointers 2-4
Program customization 14-16
Proprinter II printer 7-58
Protected field 6-56
PSCREEN 4-436
Pseudo-scan code 3-71
PTIME 11-64
Pulldown menus 4-297
Purchase Agreement 1-45

Q

Queues 8-19
Quick Reference Chart 1-38

Quicksort 14-1

R

Range checking (large arrays) 8-158
READ.ME 1-5
Reading/writing text files 5-19
RealField 6-36
Regular field 6-21
Reject functions (for DirList) 4-245
Replay 13-2
Reports 7-68
Required fields 6-126
Run-length compression 4-165
Runtime error recovery 14-25
Runtime errors 11-16

S

SaveInt1B 3-6
Saving, windows 4-112
Scan code 3-31
Scrap macro 13-18
Screen rectangles 4-9
Screen update routines 6-61
Screen/keyboard Handling 1-1
Scroll bars 4-21, 4-25, 4-113
Scrolling, windows 4-112
Search attribute (for files) 4-274
Search mask (for files) 4-242, 4-274
Search string header (for files) 4-209
Search-and-replace 5-1
Secondary command tables 3-90
Secondary field options 6-24
SelectField 6-17
Selecting fields 6-3, 6-58
Selecting fields, Action Routines 6-85
Semi-literal characters 6-5
Separator bar 4-328
Shadows 4-21, 24
ShortIntField 6-36
SideKick 11-21
Simple field 6-21
Simple line editor 6-197
SimpleCharEditor 6-52, 6-120, 6-179
SimpleStringEditor 6-52, 6-118, 6-177
SingleField 6-37
Singly linked list 8-29
Site licensing 1-45
Slider 4-32
SMACS 13-23
Sorting 14-45
Soundex 9-26
StackWindow 4-5
Stacks 8-6, 11-1, 11-15
Standard TSR interface 11-19
Static methods 2-4
StaticQueue 8-19
Status byte (for printing) 7-5
Stream 2-7, 8-90
Stream manager 8-94
Stream, memory 8-94
Stream, registration routines 6-50, 8-94
Strenczewilk, Don 9-5
String dictionary 8-67
String handling 1-2, 9-1
String management 8-67
String searching 4-198
StringArray 8-67
StringEditor 6-52, 6-116, 6-179
StringField 6-37
StringSet 8-67
StripPicture 6-14
Subfields 6-5, 8
Swap region 11-2
Swapping, TSR 11-2, 11-34
Swapping, disk/EMS 11-40
Swapping, exec manager 10-32
Swapping, messages 11-40
System requirements 1-4

T

Tab characters 5-20, 9-11
TaskView (see OmniView)
Technical support line 1-46
Terminate and Stay Resident 11-1
Text editor 1-2, 5-1, 5-3
Text editor, memo fields 5-37
Text file device drivers 7-8
TextAttr 3-6
TextField 6-15
ThwartSideKick 2-10
Time variables 6-206
TimeField 6-37
Toshiba P321SL or P341SL printers 7-58
TSR 11-1
TSR, unloading 11-20
Turbo Professional, converting from 15-16
TURBO3 3-3
TURBO5.MAC 13-23
Typecasting 8-31, 14-28

U

Unit dependencies 15-4
Unloading a TSR 11-20
UseAdjustableWindows 2-10
UseBCD 2-10
UseCrt 2-10
UseDates 2-10
UseHotSpots 2-10
UseMouse 2-10
UseScrollBars 2-10

UseShadows 2-10
User exit commands 3-95

V
Validation (see Fields)
VDISK.SYS 12-1
VGA 3-1, 3-7, 3-22, 3-35
Video attribute 3-1, 3-4, 3-48
Video memory 3-29
Video page 3-1, 3-24, 3-36, 3-39, 3-66, 3-79
Video segment 3-24, 4-173
Virtual methods 2-4
Virtual screen 4-173, 6-101

W
Warranty period 1-45
WFrame 4-21
White space 9-31
WindMax 3-6
WindMin 3-6
Window 4-8
Window, coordinates 3-33, 4-55
Window, headers 4-23
Window, hierarchy 4-1, 4-55, 4-138
Window, manager 4-56, 4-112
Window, mixing RawWindow and
    StackWindow 4-116
Window, moving and resizing 4-113
Window, objects 1-1, 4-1, 4-8
Window, saving 4-112
Window, scrolling 4-112
Window, separator bar 4-328
Window, shadows 4-21, 24
Window, sizing for PickLists 4-195
Window, stack 4-112, 15-15
Window, streams 4-57
Window-relative writing 4-103, 4-113
WINWOW 4-438
Wrap mode 6-62, 6-80
Word wrap 9-34
WordField 6-38

Y
YesNoField 6-38

Z
Zero-based arrays 8-158

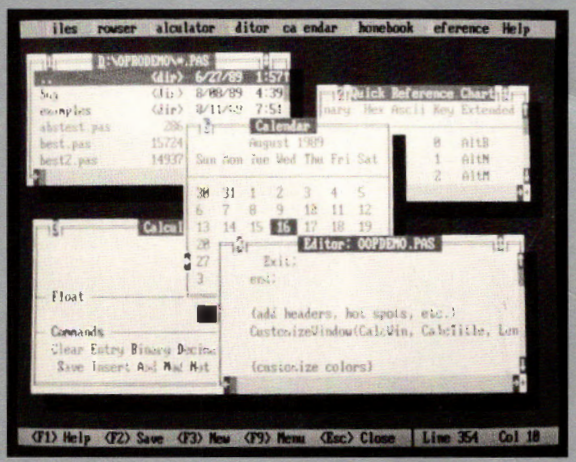
Object Professional™

A Powerful Library for Object-Oriented Programming in Pascal

Object-oriented programming multiplies your productivity by promoting reuse of proven software modules. Don't start from scratch with your objects! Use Object Professional 1.0, a huge library of over 100 powerful object types containing over 1000 methods.

Powerful User Interface Objects

The window objects let you use multiple overlapping and resizeable windows. Your windows will provide capabilities like:



- menus
- pick lists
- file browsing
- scrolling data entry screens with:
  - help capability
  - line editors
  - data validation and formatting
  - hidden and protected fields
  - matching capability for printed forms
- mouse support and scroll bars
- text editing

The window objects are completely flexible—you can create text-mode PM look-alikes or your own unique look and feel. Includes source generator for menus and entry screens.

Data Object Types . . . and System-Oriented Routines Too !

Build your programs from proven and documented object types like stacks, linked lists, virtual arrays, and more. System-oriented routines provide: ■ Swappable TSRs in only 6K of RAM ■ Interrupt management ■ EMS management ■ DOS and BIOS capabilities, and more.

Documentation: Clear and Complete

Object Professional includes clear, comprehensive documentation, on-line help, full source code, technical support, and hot demo programs. You pay no royalties when you use these routines.

Object Professional requires
Turbo Pascal 6.0 or 5.5, PC-DOS or
MS-DOS 2.0 or later, and an IBM PC,
XT, AT, PS/2 or compatible.

TURBO Power

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

©TurboPower Software 1991